Snap and Snack: A Greedy Approach to Tuning a Food Classification Transfer Learning Task

Cristian Lomeli, Andrea Dahl, Kevin Tien
Department of Computer Science
Stanford University
Stanford, CA 94305
{clomeli, ahdahl, kttien}@cs.stanford.edu

Abstract
Hyperparameter tuning is a big time-intensive part of training any Deep Learning model. For our food image classification task, we utilized an existing generic model and customized it for our specific task. In this transfer learning project, we took a greedy search approach to testing hyperparameters, the most important of which was how many layers to defreeze and subsequently train ourselves. We were able to outperform models initialized with random hyperparameters with our greedy search for effective hyperparameters.

Introduction
With thousands of possible hyperparameter combinations for any deep learning task it is very difficult to find the exact set of hyperparameters to best accompany your deep learning model and specific dataset. Thus we tested a greedy search approach for finding effective hyperparameters faster than other methods of hyperparameter tuning. The specific task we decided to focus on was a food image classification task. Presented an image of a prepared/cooked food, the CNN classifies the food by its respective name. We chose this specific problem as we thought it would be an interesting difficult task considering the vast variety of different foods, as well as the great similarities between similar food types.

Related Work
This project utilizes a pre-trained image classification model, ResNet50 [3]. ResNet introduced "identity shortcut connections" that skip one or more layers. Stacking layers doesn't degrade network performance, because stacking identity mappings on the current network results in architecture that performs the same. Using this preexisting architecture as a base for transfer learning, we improve upon its performance in a custom search method. The most common methods for hyperparameter tuning search are random and grid. Grid search builds a model for every combination of hyperparameters specified and evaluates each model. A more efficient technique is the randomized search, where random combinations of the hyperparameters are used to find the best solution. Our chosen method for hyperparameter search is a greedy search method similar to Kohavi and John's [6], which uses best-first search and cross-validation to explore the space of parameter values, running the basic algorithm many times on training and holdout sets produced by cross-validation to get an estimate of the expected error of each parameter setting.

Code Github URL: https://github.com/ahdahl/SnapAndSnack

Dataset details
We experimented with two data sets both sourced from a German social media outlet for food called chefkoch.de. We used Muriz Serifovic's data scraper [1] to obtain the images and their labels. The first data set we experimented with consisted of 11,035 food images from 390 different food classes, and our final data set was composed of 20,454 images from 184 different food classes. When using the smaller data-set with our image classification model we were not able to achieve a decent f1 score with any model due to the class imbalances in the data-set. Most classes had between 10 and 20 images and a few had over 200 images. Our newer data set only includes classes with between 100-125 images each.

Methods

4.1 Loss Formulation

Our task is a multi-class classification problem, thus we chose to use Categorical Cross-Entropy Loss [5] to optimize our model. Its formula is as follows:

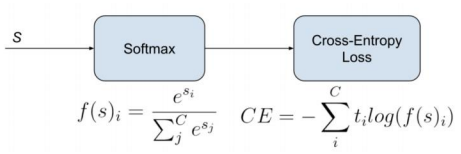

Figure 1: Categorical Cross-entropy Loss

where C is the number of classes, s_i is the calculated softmax score for the ground truth class, s_j is the softmax score for all other classes, and t_i is a binary indicator to ensure loss is calculated only for the class of interest at the time. It is a standard loss function used often in multi-class image classification tasks.

4.2 Architecture

During our initial architecture search, we used our early imbalanced class data set to test the accuracies of three models pre-trained on ImageNet: VGG16 [2], Resnet50[3], and InceptionV3 [4], without training any of the convolutional layers. We sought to use the results as an initial screening for our final model, which we would then use for our final image classification task. We also experimented with concatenating the output feature vectors of the three models and training a classifier on that. Our initial results are shown in Table 1: Although our combined network performed the best, we

Model	Test Accuracy
Combined Network	.408
VGG16	.378
ResNet50	.181
InceptionV3	.167

Table 1: Untrained Model Results

abandoned this approach due to concerns of over-fitting and difficulty of tuning, as well as concerns about poor F-1 score.

Furthermore, after consulting our TA, Sarah Najmark, and reviewing the literature, we decided to proceed with the Resnet50 model because of its proven performance on image classification against models like VGG-16, as well as its faster convergence[3].

During our training and testing we kept most of the architecture of Resnet50 untouched except for adding fully connected layers, drop out layers, and a softmax output layer. We did however make modifications to the weights of the Resnet50 model by unfreezing some of the final convolutional blocks, to allow better fitting to our data-set. After allowing for trainable convolutional layers, we could see in Figure 2, that the network was able to pick up on the higher level features of each of the food items.

Figure 2: Class Activation Maps of Last Convolutional Layer for Several Food Items.

4.3 Training

We initialized all models using pre-trained weights on ImageNet and used the Adam optimization

algorithm to update weights. We chose Adam since it has been shown to meet convergence quickly and is utilized as the default optimization method for deep learning applications. While performing our greedy search for optimal hyperparameters we would train the model for 1-5 epochs at a time, iteratively evaluating the performance of the model to the test set after each set of epochs. The model trained with our approach was then compared to the results of models trained with random parameters, which were also trained for 30 epochs. Our train/test split was 75/25 giving us 5,025 images in the test set and 15,347 images in the train set, each with resolution 420x280x3. We perform data augmentation on the training set during training using Keras' Image Data Generator function, which applies random rotation(within 60 degrees) and horizontal flip transformations to the images before training. We did this to try and reduce any over-fitting and generalize better to our data set.

4.4 Evaluation Metric

As our overall evaluation metric and to each class, we apply the macro F1 score, which is the harmonic mean of precision and recall:

$$\mathbf{F1} = \frac{2}{\frac{1}{precision} + \frac{1}{recall}}$$

Compared to raw accuracy, F1 score provides a better description of the mechanisms of the model's classification, allowing us to analyze why certain classes may be outperforming others, especially given our initial data imbalance.

Experiments/Results/Discussion

5.1 Hyperparameter Tuning Strategy

We approached hyperparameter tuning with a greedy search strategy of adjusting parameters iteratively throughout a models training process. This is done by testing different sets of hyperparameters thoughout different 1-5 epoch intervals. For example for the first epoch we tried out 3 different learning rates to train with then test on the same dataset. We then chose the learning rate which resulted in the highest macro f1-score(or accuracy if the macro-f1 scores were tied) and trained the model with this parameter for epochs one through 5, using the weights we just saved from the previous epoch, and move on to tuning a different hyperparameter. Figure 3 demonstrates the loss over epochs 15-20, when testing out the hyperparameter for four different trainable layer amounts.

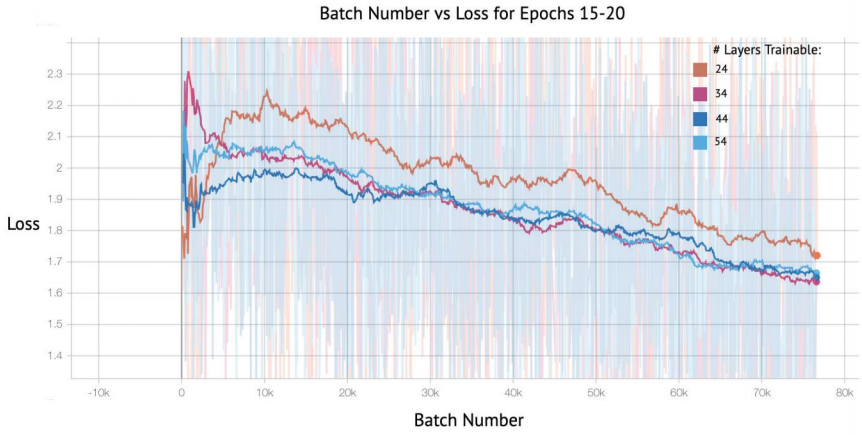

Figure 3: Batch Number vs Loss for Epochs 15-20

The full parameter search is demonstrated in Figure 4. The Flow Chart shows which set of parameters performed best throughout each set of epochs.

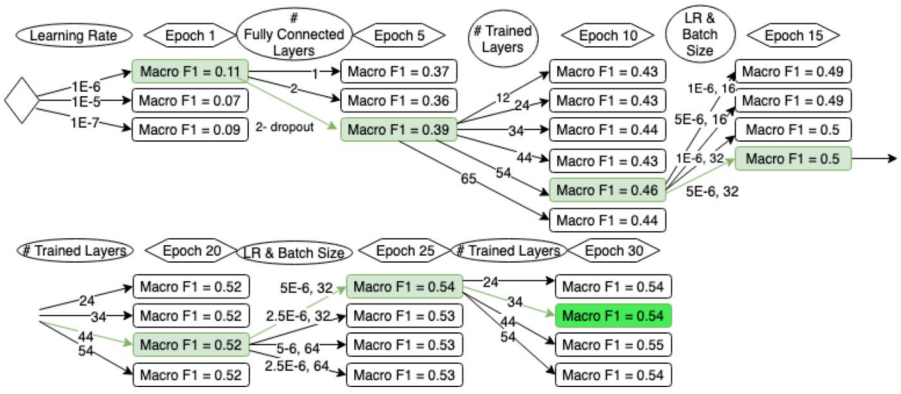

Figure 4: Hyperparameter Tuning Flowchart

The order of hyperparameter tuning is as following:

1. Epoch:0-1, Learning Rate
2. Epoch:1-5, Fully Connected Layer(s)
3. Epoch:5-10, Number of Trainable Layers
4. Epoch:10-15, Batch Size and Learning Rate
5. Epoch:15-20, Number of Trainable Layers
6. Epoch:20-25, Batch Size and Learning Rate
7. Epoch:25-30, Number of Trainable Layers

We first tuned our learning rate because of its importance in training convergence and efficiency; a learning rate that is too high will never converge, and one that is too low takes too long to train. Next, we tested our model's performance with one or two fully connected layers for four epochs, following the same process outlined above. The last fully connected layer(s) allow form more or less specialization and fitting to our image classification task. We also tested the effect of dropout between the last two connected layers, a regularization technique that can reduce over-fitting and help the model rely on more salient features. We tested this hyper-parameter early to keep the architecture of the model constant for the rest of training to make sure weights translate properly s training progressed. The results from this set of epochs suggested that having two fully connected layers with dropout would perform the best.

We utilized this paradigm next to tune the number of trainable layers in the model, and continued to alternate between readjusting the learning rate and batch size and unfreezing different numbers of trainable layers. We alternated tuning these parameters as we knew learning rate is often tuned during training (usually through decay), number of trainable layers is very important to transfer learning, and batch size is also important to adjust throughout training [7].

5.2 Results

We compare our model to the performance of four models initialized with random hyperparameter values also trained for 30 epochs. We chose four since takes the same amount of time to test as it did for us to go through our greedy parameter search. The results are reported in Table 2.

Model and Hyper-Parameters	F1-Score
Model and Hyper-Parameters from our Greedy Approach	**.55**
lr=.00001, 2fc, 54 trainable, batch = 32	.54
lr=.00001, 1fc, 44 trainable, batch = 16	.54
lr=.000005, 2fc-dropout, 54 trainable, batch = 16	.54
lr=.00005, 2fc, 64 trainable, batch = 32	.53

Table 2: Food Image Classification Results vs Randomly Initialized Hyper-Parameters

5.2 Error Analysis

Evaluating the performance of our model on each of the food classes, we found that the food items with the highest micro-f1 were unique food items with unique textures. These classes included foods like Watermelon Shark(.96), Sushi(.86), and Waffles(.84). And the models perform poorly on foods like Lemon Cake(.23), Orange Cake(.00), Eggnog Cake(.09) where discrepancies between flavors and styles might be much harder to distinguish and where the overall appearance of that type of food are extremely similar. Example images of each food mentioned are pictured in Figure 5.

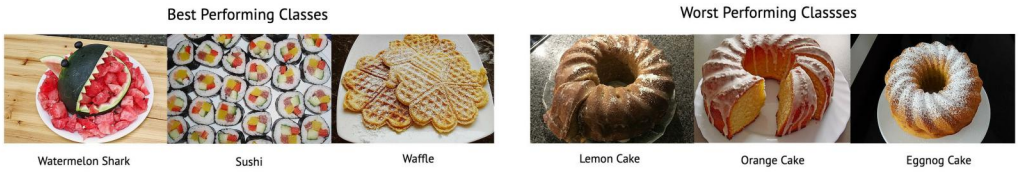

Figure 5: Best and Worst Performing Foods.

Conclusion/Future Work

Our highest performing model was the model trained using our greedy hyperparameter tuning method. It used 2 fully connected layers with dropout before the output layer, switched between 54, 44, and 33 trainable layers, switched between learning rate of 1e-5 and 5e-6, and a batch size 32. We think this is the case as it our method allows for hyperparameters to adapt to different values during the different phases of the training process. It outperformed randomly initializing hyperparameters given the same amount of training time.

As for future work there are a lot of possible modifications to our greedy algorithm that may improve it. Some logical next steps could involve experimentation with different hyperparameters to tune, as well as altering the number of epochs to train in each round. We could also apply the model to other cuisines of food, as the dataset of German foods was heavily concentrated in pastries and dessert foods. Although we used some reflections and rotations to augment the dataset, further modifications could include the addition of noise to the images to make the model more robust to imperfect images. In particular, this could help the model perform better in a possible application to real-time food identification in which the images are taken quickly rather than for a social media outlet. It would also be interesting to see if this tuning method also generalizes to other image classification or deep learning tasks.

References

[1] Mezgec S., Korousic Seljak B. NutriNet: A Deep Learning Food and Drink Image Recognition System for Dietary Assessment. Nutrients. 2017;9:657. doi: 10.3390/nu9070657.

[2]Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

[3] He, Kaiming, et al. “Deep Residual Learning for Image Recognition.” 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, doi:10.1109/cvpr.2016.90.

[4] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, Zbigniew Wojna. "Rethinking the Inception Architecture for Computer Vision." IEEE Transactions on Pattern Analysis & Machine Intelligence 7 arXiv:1512.00567 (2015)

[5] https://gombru.github.io/2018/05/23/cross_entropy_loss/

[6] Kohavi, R., John, G.: Automatic Parameter Selection by Minimizing Estimated Error. In: Prieditis, A., Russell, S. (eds.) Proceedings of the Twelfth International Conference on Machine Learning, pp. 304–312. Morgan Kaufmann Publishers (1995)

[7] Smith, Samuel L., Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. "Don’t decay the learning rate, increase the batch size." arXiv preprint arXiv:1711.00489 (2017).