SUPERVISED LEARNING - FOOD CLASSIFICATION

Mirko Morello
920601
m.morello11@campus.unimib.it

Andrea Borghesi
916202
a.borghesi@campus.unimib.it

January 5, 2025

1 INTRODUCTION

The goal of this project is to address an image classification task using a food dataset, while operating under certain constraints. We will explore multiple strategies, all grounded in Convolutional Neural Networks (CNNs), to tackle this challenge. One approach will involve proposing our own CNN architecture called TinyNet, designed with the constraint of utilizing fewer than one million parameters. Additionally, we will leverage self-supervised learning techniques and hyperparameter tuning to maximize the achievable accuracy within the given limitations.

2 DATASET

2.1 Overview

The dataset utilized in this study consists of a total of 158,846 RGB images of food with varying size, divided into three distinct subsets to facilitate the training, validation, and testing phases of the supervised learning model. The composition of the dataset is as follows:

- **Training set:** The training set comprises 118,475 images, which constitute approximately 74.6% of the total dataset. This subset is employed to train the supervised learning model, enabling it to learn the underlying patterns and features associated with each food category.
- **Test set:** The test set consists of 28,377 images, accounting for roughly 17.9% of the dataset. This subset has no labels and has been used to perform a Self-Supervised task, which will be discussed in future sections.
- **Validation set:** The validation set contains 11,994 images, representing approximately 7.5% of the dataset. This subset is used to evaluate the performance of the trained model on previously unseen data, providing an unbiased assessment of its generalization capabilities.

Each image within the dataset is assigned to one of 251 distinct categories, which correspond to the specific type of food depicted in the image. These categories encompass a wide range of food items, ensuring a diverse and comprehensive representation of various culinary classes even if there are present more than a few instances of useless examples. The categorical labels associated with each image serve as the ground truth for the supervised learning task, enabling the model to learn the mapping between the visual features of the images and their corresponding food categories.

2.2 Data exploration

Upon examining several images, it became apparent that some did not depict food at all. Additionally, certain images presented the food item occupying a very small area, making feature extraction challenging. While an attempt could be made to clean up some of these mislabeled or problematic images, the following study aimed to address such issues.

2.3 Data augmentation

To better capture the diversity present in the image distribution, we performed slight data augmentation on the training dataset. However, to avoid altering the inherent features of each image, we refrained from applying any color transformations. Instead, we focused on random rotations, translations, flips, and affine transformations - operations that preserve the core visual attributes while introducing variation. The random augmentations applied included:

- Random horizontal and vertical flipping: helped to capture different orientations that the food items may appear in.
- Random affine transformations:
  - up to 90 degree rotations augmented the dataset with multiple rotated views of the same food item, mimicking how it can be positioned differently.
  - 5% translations accounted for cases where the food was not perfectly centered in the image.
  - 10 degree shear helped make the model robust to perspective distortions.

Ultimately, to prepare the data for neural network training, each image was resized to 128x128 pixels, a dimension dictated primarily by hardware constraints. It's important to note that these transformations were applied randomly, meaning that each time an image was extracted, the augmentation differed. Since some images tended to have the food off-center or with an extremely wide aspect ratio, we decided against random scaling. A sample transformation is depicted in Figure 1.

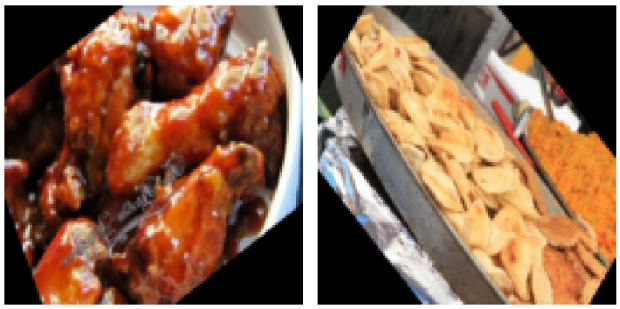

Figure 1: Two samples from the augmented dataset.

3 Metrics and Validation

The task is multiclass classification, and given that the dataset is properly stratified we decided to use micro accuracy as validation and testing metric, giving us feedback about how many correct classes we predicted over all the predictions taking into account the imbalances of the classes in the provided dataset. We will also look at the F1-score for each class, a measure that takes into consideration both precision and recall, to better analyze if our models struggled with any class in particular.

The loss function used as a criterion to assess the performances of the net during each batch is the Cross-Entropy loss, a widely used loss for multiclass classification tasks.

4 CNN Architecture - TinyNet

In this project, we designed a custom Convolutional Neural Network (CNN) architecture called "tinyNet" to tackle the food classification task. The tinyNet comprises 10 convolutional layers to handle the feature extraction, and 2 fully connected to handle the classification. Our first implementation is described in Table 1, totaling 999,675 parameters. This first implementation is completely trained from scratch.

The choice of using a Convolutional Neural Network (CNN) was primarily motivated by its ability to automatically identify and extract relevant features from the input data, which can then be passed through a fully connected classification layer. The convolutional part of the network was structured as a macro layer containing two convolutional layers with GELU (Gaussian Error Linear Unit) as the activation functions, followed by batch normalization and max pooling. This approach of stacking multiple convolutional layers with the same kernel sizes has been proven to significantly enhance the network's performance [12] [13] [7]. Stacking multiple convolutional layers increases the receptive field and allows the network to encode more complex features. The inclusion of a non-linear activation function after each layer enables the network to capture non-linear patterns in the data. Furthermore, using small kernel sizes in stacked convolutional layers instead of a single layer with a larger kernel reduces the number of parameters while retaining the aforementioned advantages. The selection of GELU as the activation function was a carefully considered choice. While LeakyReLU [9] was also a viable option, GELU was preferred due to its empirical performance in modern neural networks, where it has been observed to provide faster convergence and higher accuracies compared to other activation functions [3]. Additionally, GELU's smoothness and differentiability over its entire domain can potentially improve gradient flow during training, mitigating the vanishing gradient problem. Although GELU is computationally more expensive than LeakyReLU, the authors were willing to accept this trade-off, expecting it to add no more than half an hour to the training time. The incorporation of batch normalization in the macro layers was motivated by its numerous benefits [4]. Batch nor-

malization was introduced after several layers to mitigate the internal covariate shift, thereby stabilizing the internal distribution of each layer. Additionally, it accelerates the training process and enables the use of larger learning rates, effectively managing the instability that higher learning rates can introduce.

Lastly, the exact choice of hyperparameters such as the number of kernels for each layer and the number of units in the fully connected layers was largely inspired by other successful networks like AlexNet [7] and VGG16 [11], which tend to gradually increase the number of kernels to slowly encode more complex features. The specific values were then adjusted to stay as close as possible to the limit of the number of parameters.

4.1 Optimizers and Schedulers

After an extensive exploration of various optimizers and schedulers, we settled on employing Adam, a state-of-the-art optimizer renowned for its effectiveness across a wide range of applications [6]. To complement Adam, we selected the Cosine Annealing scheduler [8]. This combination enabled an efficient convergence, enhancing the model's ability to generalize effectively. By carefully tuning these components, we aimed to strike the right balance between exploration and exploitation, ultimately leading to improved performance and robustness in our deep learning model with a starting learning rate of 10^{-3} down to a minimum of 10^{-4}.

5 HYPERPARAMETER TUNING

To adhere to the project's constraint of maintaining the total number of parameters below 1 million, we meticulously tuned several hyperparameters of the TinyNet architecture. Specifically, we focused on optimizing the number of kernels for each convolutional layer, with the exception of the output layer, which was fixed at 32 kernels. Additionally, we fine-tuned the number of units in the classification head, totaling 6 hyperparameters to optimize. To streamline and automate this intricate optimization process, we leveraged the powerful capabilities of the Optuna library [1]. Optuna, a modern framework for hyperparameter tuning and model selection, facilitated our exploration of the hyperparameter space. We configured an Optuna study object with a 'maximize' direction, and employed a pruning strategy to terminate trials based on the total parameter count, either if it was above 1 million or below 900 thousands. This pruning mechanism ensured that our search remained focused on architectures

Layer	Details	Output Shape
Input Image: $3 \times 128 \times 128$		
Conv1	Conv2d(3, 8, 3x3, stride=1, padding=same)	$8 \times 128 \times 128$
	GELU()	$8 \times 128 \times 128$
	Conv2d(8, 32, 3x3, stride=1, padding=1)	$32 \times 128 \times 128$
	BatchNorm2d(32)	$32 \times 128 \times 128$
	GELU()	$32 \times 128 \times 128$
	MaxPool2d(2x2, stride=2)	$32 \times 64 \times 64$
Conv2	Conv2d(32, 32, 3x3, stride=1, padding=same)	$32 \times 64 \times 64$
	GELU()	$32 \times 64 \times 64$
	Conv2d(32, 64, 3x3, stride=1, padding=1)	$64 \times 64 \times 64$
	BatchNorm2d(64)	$64 \times 64 \times 64$
	GELU()	$64 \times 64 \times 64$
	MaxPool2d(2x2, stride=2)	$64 \times 32 \times 32$
Conv3	Conv2d(64, 64, 3x3, stride=1, padding=same)	$64 \times 32 \times 32$
	GELU()	$64 \times 32 \times 32$
	Conv2d(64, 128, 3x3, stride=1, padding=1)	$128 \times 32 \times 32$
	BatchNorm2d(128)	$128 \times 32 \times 32$
	GELU()	$128 \times 32 \times 32$
	MaxPool2d(2x2, stride=2)	$128 \times 16 \times 16$
Conv4	Conv2d(128, 128, 3x3, stride=1, padding=same)	$128 \times 16 \times 16$
	GELU()	$128 \times 16 \times 16$
	Conv2d(128, 172, 3x3, stride=1, padding=1)	$172 \times 16 \times 16$
	BatchNorm2d(172)	$172 \times 16 \times 16$
	GELU()	$172 \times 16 \times 16$
	MaxPool2d(2x2, stride=2)	$172 \times 8 \times 8$
Conv5	Conv2d(172, 172, 3x3, stride=1, padding=same)	$172 \times 8 \times 8$
	GELU()	$172 \times 8 \times 8$
	Conv2d(172, 32, 3x3, stride=1, padding=1)	$32 \times 8 \times 8$
	BatchNorm2d(32)	$32 \times 8 \times 8$
	GELU()	$32 \times 8 \times 8$
	MaxPool2d(2x2, stride=2)	$32 \times 4 \times 4$
FC1	Linear($32 \times 4 \times 4$, 256)	256
	Dropout(0.2)	256
	GELU()	256
FC2	Linear(256, 251)	251

Table 1: TinyNet architecture - 999,675 paramaters

that could exploit as much as possible the imposed constraint. By seamlessly integrating Optuna into our workflow, we could systematically evaluate numerous hyperparameter configurations, by iteratively testing Optuna's proposed configuration for a limited number of epochs (between 8 and 15). This data-driven approach, coupled with Optuna's sophisticated optimization algorithms, enabled us to identify the hyperparameter combination that yielded the best results while respecting the project's constraints. To speed up the process, as it takes several hours, we decided to not perform any augmentation during the optimization.

The optimized architecture, as shown in Table 2, was able to achieve an accuracy of 26.02% after just 10 epochs of training. After the optimization process, it became evident that the architecture was oscillating between two different structural approaches. The first approach involved increasing the number of kernels up to the fourth macro layer and then decreasing the number of kernels in the fifth macro layer. The second approach, which achieved 26.02% accuracy, followed the same structure as the orig-

inal TinyNet architecture but allocated more parameters to the classification head (fully connected layers).

Layer	Details	Output Shape
Input Image: $3 \times 128 \times 128$		
Conv1	Conv2d(3, 22, 3x3, stride=1, padding=same)	$22 \times 128 \times 128$
	GELU()	$22 \times 128 \times 128$
	Conv2d(22, 32, 3x3, stride=1, padding=1)	$32 \times 128 \times 128$
	BatchNorm2d(32)	$32 \times 128 \times 128$
	GELU()	$32 \times 128 \times 128$
	MaxPool2d(2x2, stride=2)	$32 \times 64 \times 64$
Conv2	Conv2d(32, 32, 3x3, stride=1, padding=same)	$32 \times 64 \times 64$
	GELU()	$32 \times 64 \times 64$
	Conv2d(32, 64, 3x3, stride=1, padding=1)	$64 \times 64 \times 64$
	BatchNorm2d(64)	$64 \times 64 \times 64$
	GELU()	$64 \times 64 \times 64$
	MaxPool2d(2x2, stride=2)	$64 \times 32 \times 32$
Conv3	Conv2d(64, 64, 3x3, stride=1, padding=same)	$64 \times 32 \times 32$
	GELU()	$64 \times 32 \times 32$
	Conv2d(64, 80, 3x3, stride=1, padding=1)	$80 \times 32 \times 32$
	BatchNorm2d(80)	$80 \times 32 \times 32$
	GELU()	$80 \times 32 \times 32$
	MaxPool2d(2x2, stride=2)	$80 \times 16 \times 16$
Conv4	Conv2d(80, 80, 3x3, stride=1, padding=same)	$80 \times 16 \times 16$
	GELU()	$80 \times 16 \times 16$
	Conv2d(80, 172, 3x3, stride=1, padding=1)	$172 \times 16 \times 16$
	BatchNorm2d(172)	$172 \times 16 \times 16$
	GELU()	$172 \times 16 \times 16$
	MaxPool2d(2x2, stride=2)	$172 \times 8 \times 8$
Conv5	Conv2d(172, 172, 3x3, stride=1, padding=same)	$172 \times 8 \times 8$
	GELU()	$172 \times 8 \times 8$
	Conv2d(172, 32, 3x3, stride=1, padding=1)	$32 \times 8 \times 8$
	BatchNorm2d(32)	$32 \times 8 \times 8$
	GELU()	$32 \times 8 \times 8$
	MaxPool2d(2x2, stride=2)	$32 \times 4 \times 4$
FC1	Linear($32 \times 4 \times 4$, 500)	500
	Dropout(0.2)	500
	GELU()	500
FC2	Linear(500, 251)	251

Table 2: TinyNet architecture tuned with Optuna optimization framework - 997763 parameters

6 Self-Supervised Learning

Self-supervised learning is an emerging paradigm in machine learning that aims to leverage the inherent structure and patterns within unlabeled data to learn meaningful representations. Unlike traditional supervised learning, which relies on explicitly labeled data, self-supervised learning enables the model to learn from the data itself by creating its own supervisory signals [5]. In the context of computer vision, self-supervised learning has gained significant attention due to its ability to learn robust visual representations without the need for extensive manual annotation. One popular approach to self-supervised learning in the visual domain is the use of pretext tasks, which are designed to encourage the model to capture relevant features and structures from the images [2].

For the specific case of the food dataset used in this study, we employ a self-supervised learning technique based on image reconstruction using black boxes. This approach involves randomly masking out portions of the input images with black boxes and training the model to reconstruct the original images based on the available context. By learning to fill in the missing regions, the model is forced to develop a deep understanding of the visual characteristics and patterns associated with different food categories.

The network, inspired by the U-Net architecture [10], has been trained for 60 epochs with the entirety of the training set and the test set, the black boxes were inserted randomly, an example of the reconstruction capabilities of this model can be seen in Figure 2.

The image reconstruction task serves as a pretext task that enables the model to learn meaningful representations of the food images without relying on explicit category labels.

The encoding part of this model is identical to the classifier architecture represented in Table 1, with the last 2 linear layers trimmed out. The decoder is a mirroring of the encoder which handles the upconvolution, it works its way up until the image has been restored to its original dimensions.

After the self-supervised pretraining stage, we employ a transfer learning approach to leverage the learned representations for the classification task. We utilize the encoder part of the pre-trained model, which is responsible for extracting meaningful features from the input images, and transfer its weights to the classification model.

This transfer learning approach allows the classification model to converge faster and achieve better performance compared to training from scratch, as it leverages the powerful features extracted by the pre-trained encoder, as can be seen in picture 2.

The combination of self-supervised pretraining and transfer learning seems to help the model to generalize better to unseen food images, as it is able to converge quicker and in a more stable manner, although its overall increase in performance is very subtle.

7 Results

The hyperparameter-tuned TinyNet, although it converged faster for the first epochs, it reached a worse plateau compared to the untuned network (Figure 3) , both with and without using the SSL weights, achieving a val-

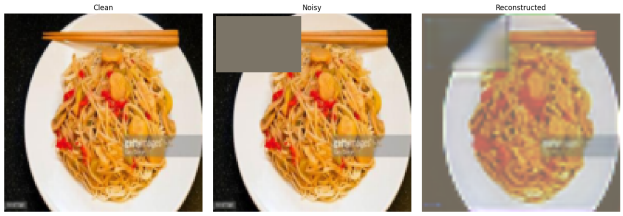

Figure 2: Stages of applying a black box and trying to reconstruct it.

idation accuracy of 43.83% without SSL pretraining and 43.93% with SSL pretraining. Although this results are not our best, they come very close to them as we will see. Its untuned counterpart (without SSL pretrained) on the other hand, achieved an accuracy of 45.31%. The hyperparameter-tuned network's performance can be attributed to the way hyperparameter tuning was conducted. Due to the time-consuming nature of this process, each configuration of parameters was tested for only 10-15 epochs. While this provided an indication of the potentially best configuration, it did not guarantee that the network's performance would plateau at that point. Nevertheless, the results suggested that the initial choice of configuration was reasonably good.

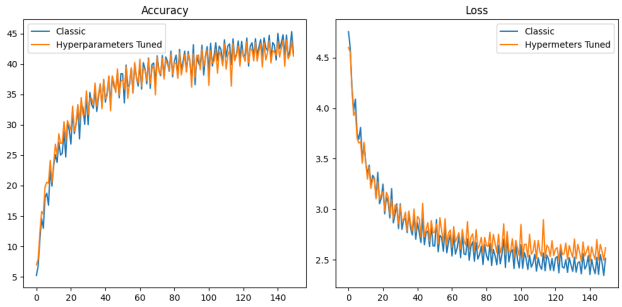

Figure 3: Accuracies of the validation and loss of the network trained with and without hyperparameter tuned, both are pretrained on the respective SSL.

Our best network, which resulted from the non-hyperparameter-tuned TinyNet (aka Classic tinyNet) initialized with the weights of the encoder of the SSL model, achieving an accuracy of 45.33% (Figure 4).
While this percentage may seem low, it is important to consider the presence of 251 different classes and the several constraints imposed for this task. Nonetheless the false positives that the model found are reasonable, as it finds hard to discern similar food such as different kind of spaghetti dishes. To account for this, we decided to compute the F1-score for each class and average it across the number of classes, resulting in a micro average F1-score of 0.4533.

When comparing the performance of the Classic tinyNet with its pre-trained version that used self-supervised learning (SSL) weights, we observed that although the pre-trained version demonstrated improved stability and faster convergence in the initial few epochs, the overall (micro) accuracy did not change significantly, with only a 0.02% improvement. This behavior can be attributed to the fact that we likely provided sufficient training epochs for both instances to converge and achieve nearly identical performance levels.

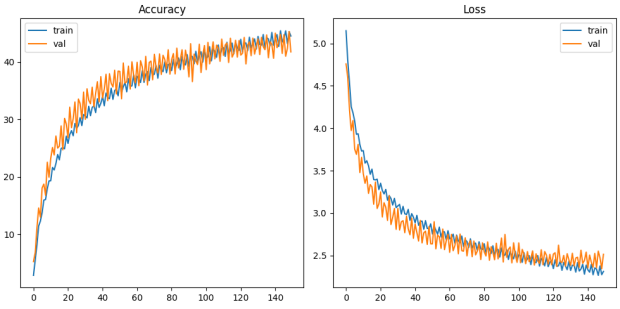

Figure 4: Accuracies of the validation and loss of the network trained with SSL weights without hyperparameter tuning.

Figure 4 shows the validation accuracies and losses of the network trained with SSL weights. The graphs do not indicate any signs of overfitting, suggesting that a more exhaustive fine-tuning could potentially lead to better results. However, due to time constraints, training for 150 epochs was sufficient to demonstrate the strength of our approach and the effectiveness of the network.

As shown in Figure 5, the network initialized with the SSL's weights achieved better accuracies faster compared to the randomly initialized network. However, likely due to the simplicity of the network, both converged to similar results. This convergence could also be attributed to the limited capacity of the network to fully exploit the benefits of the SSL pre-training. Despite this, the results still slightly leverage the SSL's transferred weights, granting a more stable and marginally faster convergence.

Although it's challenging to visualize the performance across the vast number of categories, a heatmap representation of the network's results can be seen in Figure 6 showing a clear diagonal with some false positives.

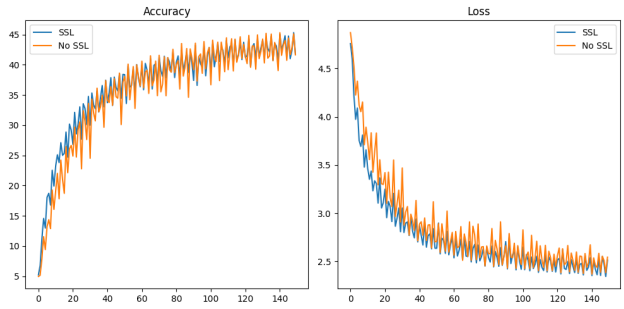

Figure 5: Validation accuracies and losses of the untuned network initialized with SSL weights and without.

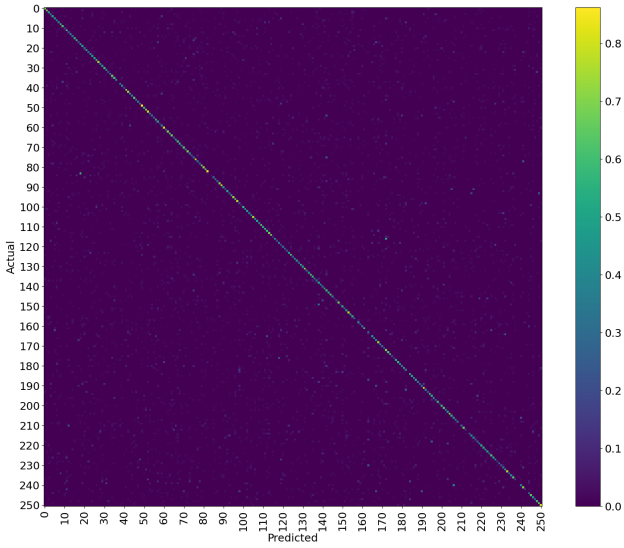

Figure 6: Heatmap of the accuracies across all the food categories for the Classic tinyNet pretrained with SSL weights.

7.1 Prediction's analysis

Our predictive models often struggle when attempting to distinguish between similar types of food. Figure 7 displays the F1 score for each food category, highlighting the classes that are most challenging to predict accurately as well as those that are relatively distinct and easier to identify correctly. While the scores appear fairly consistent across most categories, certain classes stand out as being exceptionally difficult to predict.

8 Conclusions

In this study, we investigated the application of Self-Supervised Learning (SSL) for food classification using the TinyNet architecture. The main objectives were to design an optimized CNN architecture, employ SSL to learn visual features from unlabeled data, and evaluate the impact of SSL on the classification performance.
The TinyNet architecture, demonstrated its effectiveness in food classification while adhering to the parameter constraint of 1 million. The SSL technique based on image reconstruction with black boxes successfully learned meaningful representations from the unlabeled food images. By transferring the learned representations to the TinyNet model, we observed a slight improvement in classification performance and a faster convergence compared to training from scratch.

Disclaimer

This report does not contain any form of plagiarism, including content generated or suggested by AI tools such as ChatGPT or similar services. All sources used have been properly cited and referenced.

References

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[2] Carl Doersch, Abhinav Gupta, and Alexei A Efros. Unsupervised visual representation learning by context prediction. In *Proceedings of the IEEE international conference on computer vision*, pages 1422–1430, 2015.

[3] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2023.

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

[5] Longlong Jing and Yingli Tian. Self-supervised visual feature learning with deep neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(11):4037–4058, 2020.

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

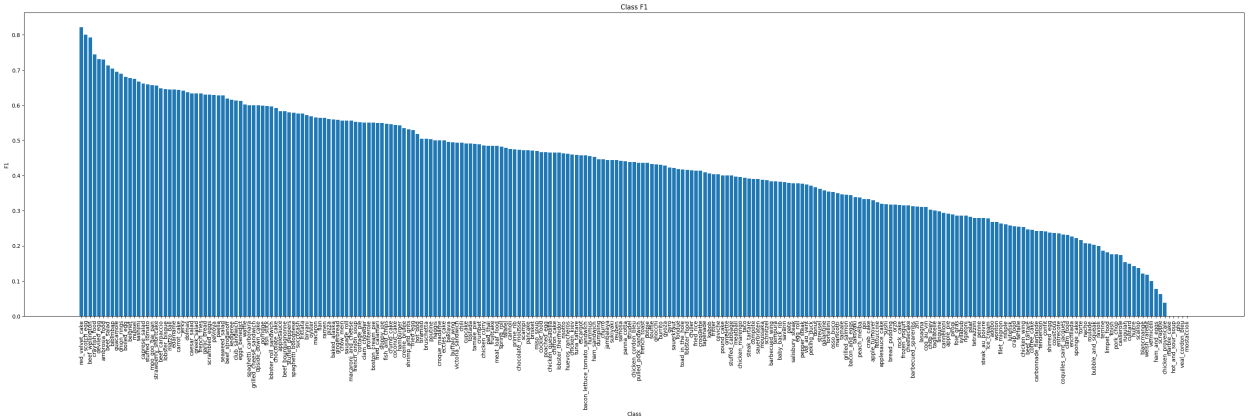

Figure 7: Sorted F1-score of each class for the Classic tinyNet pretrained with SSL weights.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[8] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[9] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 3–8, 2013.

[10] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

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

[12] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[13] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.