Convolutional Neural Networks for Food Classification

May 2025 – Group 12

First Name	Last Name	Student Number
Nadia	Koghee	s3992241
Nicolas	Ramos Fernandez	s3917886
Anna Michelle	van der Spek	s4077644
Adrien Joon-Ha	Im	s3984389

Chapter 1

report

1.1 Introduction

In this project, we design and implement a Convolutional Neural Network (CNN) to classify food into 91 categories. This project aims to create a recommendation system that deduces the user's food preferences based on 10 images uploaded by the user. To develop the model, we used a provided dataset comprising approximately 45,000 training images and 22,000 test images, which are classified based on their type. This project is further extended by using an LLM to generate a description of the food preference profile of a certain user based only on the 10 images they provide.

1.2 Implementation

1.2.1 Data Preparation

The dataset used in this project consists of approximately 67,000 labeled images divided into a training set (45,000) and a testing set (22,000). These images are organized into class-specific directories: `train/` and `test/`. Each image is also classified under each directory under one of 91 predefined food categories. The images were loaded and labeled using `torchvision.datasets.ImageFolder`. In order to prepare the data for training, we defined two transformation pipelines with `torchvision.transforms`. For the training set, we applied several augmentations: `RandomResizedCrop(224)`, `RandomHorizontalFlip()`, `RandomRotation(15)`, and `ColorJitter`, which allow us to crop the images randomly and vary brightness, contrast, saturation and hue. The images were then converted to tensors and normalized [1]. For the test and validation tests, we similarly applied with `Resize((224,224))`, which was followed by tensor conversion and the same normalization as in the training set.

We also split the training set into an 80/20 train-validation split using `torch.utils.data.random_split`, and loaded the data into memory using PyTorch's `DataLoader` with a batch size of 64, shuffling enabled, and `pin_memory = True` to optimize data transfer to the GPU when it is available. Additionally, we set a global seed of 42 to ensure reproducibility.

1.2.2 CNN Model Architecture

As for the implementation of our `FoodCNN` model, we considered two options: the first model was the Inception Model V2 GoogleNet, and the second was the AlexNet.

Our inception model's architecture was mainly inspired by the second version of the model, with additional modifications of the first inception model, GoogleNet. The model was modified to be properly adjusted to our project. In the model, four main branches processed the input in parallel. Each of these branches extracts a different set of features, and concatenating the results provides the model with a detailed set of the similarities between images of a class (category). However, a problem with the inception model V1 is that the convolution layers cause the input dimensions to decrease by a large margin, which is then reflected in the accuracy [2]. The inception model V2 tries to reduce this problem by using two 3*3 convolutions to make the model more efficient regarding accuracy and speed[3].

Thus, after working with both models, and only being able to reach an accuracy of 37,08% on the Inception model, we were compelled to continue with the AlexNet model as we had computed a higher accuracy on the

test dataset. In our implementation of the AlexNet model, the core principles were conserved: a deep stack of convolutional layers interleaved with non-linear activations and pooling [4, 5, 6]. However, the architecture was modified to accommodate the food dataset, consisting of 91 classes. As with the Inception model, there were a significant number of tries before exceeding 50% accuracy on the test data.

The two prior changes were the addition of an SE block[7] and the use of SiLu()[8]. The SE block was implemented to improve feature selection, however, we believe it did not work as well with our dataset due to the variety of classes the CNN had to learn. Hence, it was more difficult to isolate which features can be used to differentiate classes. The SiLu activation seemed to provide a similar accuracy to ReLu for our case, so we stuck with ReLu().

The final implemented model consisted of two main components: a convolutional feature extractor and a fully connected stack module.

The feature extractor consists of seven convolutional layers with varying channel dimensions, as opposed to the five convolutional layers in the original network. It starts with a 7 by 7 convolution and continues through a series of 3 by 3 convolutions. The ReLU activation functions follow each layer, and batch normalization is applied throughout the layers in order to stabilize the training and accelerate convergence. After some layers, max-pooling is applied to reduce spatial dimensionality. Finally, the last convolutional block is followed by a global average pooling layer, which compresses the spatial resolution to one dimension.

The extracted features are then passed to a classification comprising a dropout layer with a probability of 0.3, and a fully connected linear layer that maps the 512-dimensional feature vector to a 91-dimensional output representing the different food classes. The original network had three fully connected layers rather than one. The linear layer is initialized using Kaiming initialization in order to train more effectively with ReLU activations. This architecture was selected to balance the complexity of the model and performance. It stays within computational constraints while solving the task effectively.

1.2.3 Training Procedure

In order to train the model, we use the AdamW optimizer with a learning rate of 0.001 and weight decay of 0.0005. AdamW differs from the Adam optimizer by removing weight decay from the gradient update of the learning curve. Instead, after the parameters have been updated and the model has been adjusted to identify more features. [9]. This improves the overall learning. The learning rate decay was done with a cosine annealing scheduler using the maximum number (40) of epochs. Cosine annealing decays the learning rate based on the cosine curve and does not reset it until the last epoch for training has been reached. This permits the model to make modifications for refinement later on, instead of decaying by a fixed factor, which could ordinarily result in a model not learning enough at later epochs.[10] The loss function used during training was Focal Loss, a modification to cross-entropy loss. This loss function allows us to manage large class imbalances found in dense object detection problems [11]. The γ, also considered as the "focusing parameter" [12] or relaxation parameter [13], which controls how much the loss calculation focuses on difficult examples. It performs best at value two [12].

The training was conducted over 40 epochs. The accuracy of the training set improved from 4.60% in the first epoch to 61.74% by the final epoch. Model checkpoints were saved after each epoch, and the best-performing model was separately stored in the file `best_model.pth`. All training runs were performed utilizing GPU acceleration when it was available.

1.2.4 Calculating Model Performance

To quantitatively calculate the model's performance, we implemented an evaluation function `calculate_accuracy` which accepts a PyTorch `DataLoader`, the trained model and the loss function used during training as inputs. The model is first set to evaluation mode, and the function is run with `torch.no_grad()` to ensure that no gradients are computed during test mode, and to reduce unnecessary gradient computations and memory usage for tensors with `requires_grad = True`. For each batch, the function computes the predictions. At the end of the loop, the loss is averaged and the accuracy is computed as the percentage of correct predictions over the total number of samples. This function was used to evaluate our final trained model on the test set.

1.3 Simulation of random user

A user of this model would provide several images for it to classify. To simulate this process, we developed a program that gathers 10 random images from the test dataset and duplicates them into a directory called

uploaded_images. Subsequently, we created a function, *make_prediction()*, that opens a directory and predicts the classes of the images stored inside. Finally, we implemented code to obtain the predicted class frequency distribution of the model's predictions across the 10 images. We observed that each predicted class appeared only once; this is possibly a result of a high number of potential classes to predict. However, if we run the simulation 30 times and sum the frequencies over all runs, we observe that some classes occur significantly more often than others; for example, chicken curry, bibimbap, and cheese plate occurred 7 times, while other foods, such as falafel, only occurred once. This may be due to the model selecting some classes more commonly than others.

1.4 Bonus: LLM-Generated Food Profile

In the last stage of our project, we simulate a real-world use of the model where a user uploads ten images of food. These images are passed through our trained `FoodCNN` model to predict the respective food categories. To prepare the images, we used the *make_prediction()* function created previously to predict the images in the *uploaded_images* directory. The model is loaded from the saved weights (`llm_images/` and is run in evaluation mode. For each image, the model outputs a predicted class, and is mapped to the corresponding food label using the original training class order.

In order to make the user experience more engaging, we also integrated a Large Language Model (LLM) to generate a personalized description of the user's food preferences. We used the Gemini API due to the availability of a free plan as well as the high observed quality of its generated tests for the given task. Once the predicted labels are obtained, they are embedded in a natural language prompt. This prompt is passed to Google's Gemini API (`genai.Client`), which generates the personalized food profile. We designed two prompts: one generating a professional and neutral analysis of the user's food profile, and another that plays the role of an Italian chef B.

1.5 Conclusion

In this project, we developed a Convolutional Neural Network to classify food into 91 predefined categories, using a modified architecture inspired by AlexNet. The model was trained without pre-trained weights and achieved a test accuracy of 52.35% after 40 epochs. To solve class imbalance, we implemented Focal Loss and used techniques such as Batch Normalization, Kaiming initialization, and the AdamW optimizer with a cosine annealing learning rate scheduler. We were also able to apply the model to a real-world application by integrating it into a food recommendation system. By using the predicted labels of ten user-uploaded images as input to a Large Language Model, we generated a personalized food profile.

In conclusion, this project demonstrates the possibility of constructing and training deep learning models from the ground up, drawing inspiration from previous studies to suggest model designs and enhance the ability to perform complex classification tasks.

Appendix A

Epochs - formatted data

```
Epoch [1/40] Test loss: 3.9818, test set accuracy 6.89%,
Train set loss: 3.7948, train set accuracy: 4.60%,
Validation set loss: 3.9732, validation set accuracy: 6.77%

Epoch [2/40] Test set loss: 3.7638, test set accuracy 10.02%,
Train set loss: 3.9352, train set accuracy: 7.26%,
Validation set loss: 3.8755, validation set accuracy: 8.20%

Epoch [3/40] Test loss: 3.5693, test set accuracy 12.39%,
Train set loss: 4.2767, train set accuracy: 10.16%,
Validation set loss: 3.7268, validation set accuracy: 10.29%

Epoch [4/40] Test loss: 3.3065, test set accuracy 16.39%,
Train set loss: 3.7774, train set accuracy: 13.58%,
Validation set loss: 3.4496, validation set accuracy: 14.27%

Epoch [5/40] Test loss: 2.9587, test set accuracy 22.50%,
Train set loss: 3.3772, train set accuracy: 17.42%,
Validation set loss: 3.2279, validation set accuracy: 18.26%

Epoch [6/40] Test loss: 2.7927, test set accuracy 25.29%,
Train set loss: 3.0325, train set accuracy: 20.90%,
Validation set loss: 3.0591, validation set accuracy: 20.75%

Epoch [7/40] Test loss: 2.6068, test set accuracy 28.97%,
Train set loss: 3.2937, train set accuracy: 24.09%,
Validation set loss: 2.8511, validation set accuracy: 25.65%

Epoch [8/40] Test loss: 2.4482, test set accuracy 32.39%,
Train set loss: 2.1825, train set accuracy: 26.95%,
Validation set loss: 2.7823, validation set accuracy: 26.67%

Epoch [9/40] Test loss: 2.3306, test set accuracy 34.40%,
Train set loss: 2.4300, train set accuracy: 29.25%,
Validation set loss: 2.6304, validation set accuracy: 29.60%

Epoch [10/40] Test loss: 2.2159, test set accuracy 36.91%,
Train set loss: 2.4514, train set accuracy: 31.35%,
Validation set loss: 2.5236, validation set accuracy: 31.58%

Epoch [11/40] Test loss: 2.1843, test set accuracy 37.78%,
Train set loss: 1.9791, train set accuracy: 33.78%,
Validation set loss: 2.4769, validation set accuracy: 32.65%

Epoch [12/40] Test loss: 2.1072, test set accuracy 39.55%,
```

```
Train set loss: 2.8305, train set accuracy: 35.39%,
Validation set loss: 2.4826, validation set accuracy: 32.81%

Epoch [13/40] Test loss: 2.0821, test set accuracy 39.79%,
Train set loss: 1.8734, train set accuracy: 37.21%,
Validation set loss: 2.4272, validation set accuracy: 33.74%

Epoch [14/40] Test loss: 1.9965, test set accuracy 41.81%,
Train set loss: 2.0331, train set accuracy: 38.38%,
Validation set loss: 2.3499, validation set accuracy: 36.03%

Epoch [15/40] Test loss: 1.9701, test set accuracy 42.63%,
Train set loss: 2.2124, train set accuracy: 40.35%,
Validation set loss: 2.3310, validation set accuracy: 35.94%

Epoch [16/40] Test loss: 1.8928, test set accuracy 44.23%,
Train set loss: 1.8597, train set accuracy: 41.74%,
Validation set loss: 2.2621, validation set accuracy: 36.98%

Epoch [17/40] Test loss: 1.8738, test set accuracy 44.95%,
Train set loss: 2.1257, train set accuracy: 42.99%,
Validation set loss: 2.2325, validation set accuracy: 38.32%

Epoch [18/40] Test loss: 1.8445, test set accuracy 46.10%,
Train set loss: 2.2262, train set accuracy: 44.56%,
Validation set loss: 2.1900, validation set accuracy: 39.36%

Epoch [19/40] Test loss: 1.8051, test set accuracy 46.75%,
Train set loss: 1.4954, train set accuracy: 45.56%,
Validation set loss: 2.1600, validation set accuracy: 39.56%

Epoch [20/40] Test loss: 1.8040, test set accuracy 47.13%,
Train set loss: 1.9521, train set accuracy: 46.87%,
Validation set loss: 2.1624, validation set accuracy: 39.91%

Epoch [21/40] Test loss: 1.7926, test set accuracy 47.60%,
Train set loss: 1.5743, train set accuracy: 48.11%,
Validation set loss: 2.0936, validation set accuracy: 41.84%

Epoch [22/40] Test loss: 1.7658, test set accuracy 47.76%,
Train set loss: 1.8694, train set accuracy: 49.17%,
Validation set loss: 2.1394, validation set accuracy: 40.95%

Epoch [23/40] Test loss: 1.7738, test set accuracy 47.77%,
Train set loss: 1.7535, train set accuracy: 50.45%,
Validation set loss: 2.0871, validation set accuracy: 41.88%

Epoch [24/40] Test loss: 1.7213, test set accuracy 48.75%,
Train set loss: 2.0632, train set accuracy: 51.34%,
Validation set loss: 2.0599, validation set accuracy: 42.05%

Epoch [25/40] Test loss: 1.7129, test set accuracy 49.20%,
Train set loss: 1.1588, train set accuracy: 52.46%,
Validation set loss: 2.0511, validation set accuracy: 42.97%

Epoch [26/40] Test loss: 1.7015, test set accuracy 49.66%,
Train set loss: 1.1307, train set accuracy: 53.52%,
Validation set loss: 2.0603, validation set accuracy: 43.31%

Epoch [27/40] Test loss: 1.6790, test set accuracy 50.20%,
Train set loss: 1.3509, train set accuracy: 54.33%,
Validation set loss: 2.0361, validation set accuracy: 43.29%
```

```
Epoch [28/40] Test loss: 1.6631, test set accuracy 50.81%,
Train set loss: 1.8097, train set accuracy: 55.63%,
Validation set loss: 2.0021, validation set accuracy: 43.65%

Epoch [29/40] Test loss: 1.6764, test set accuracy 50.44%,
Train set loss: 1.5321, train set accuracy: 56.63%,
Validation set loss: 2.0351, validation set accuracy: 43.67%

Epoch [30/40] Test loss: 1.6803, test set accuracy 50.76%,
Train set loss: 1.3835, train set accuracy: 57.54%,
Validation set loss: 1.9763, validation set accuracy: 44.76%

Epoch [31/40] Test loss: 1.6607, test set accuracy 51.32%,
Train set loss: 1.0589, train set accuracy: 58.32%,
Validation set loss: 1.9560, validation set accuracy: 45.48%

Epoch [32/40] Test loss: 1.6497, test set accuracy 51.78%,
Train set loss: 1.2262, train set accuracy: 58.97%,
Validation set loss: 1.9768, validation set accuracy: 44.58%

Epoch [33/40] Test loss: 1.6645, test set accuracy 51.68%,
Train set loss: 0.9578, train set accuracy: 59.65%,
Validation set loss: 2.0036, validation set accuracy: 44.57%

Epoch [34/40] Test loss: 1.6459, test set accuracy 51.88%,
Train set loss: 1.6436, train set accuracy: 60.21%,
Validation set loss: 1.9889, validation set accuracy: 45.28%

Epoch [35/40] Test loss: 1.6542, test set accuracy 52.01%,
Train set loss: 1.0175, train set accuracy: 60.68%,
Validation set loss: 1.9619, validation set accuracy: 45.17%

Epoch [36/40] Test loss: 1.6502, test set accuracy 51.96%,
Train set loss: 1.0450, train set accuracy: 61.22%,
Validation set loss: 1.9666, validation set accuracy: 45.65%

Epoch [37/40] Test loss: 1.6416, test set accuracy 52.20%,
Train set loss: 1.1446, train set accuracy: 61.42%,
Validation set loss: 1.9579, validation set accuracy: 46.00%

Epoch [38/40] Test loss: 1.6482, test set accuracy 52.28%,
Train set loss: 1.0417, train set accuracy: 61.71%,
Validation set loss: 1.9618, validation set accuracy: 45.62%

Epoch [39/40] Test loss: 1.6448, test set accuracy 52.24%,
Train set loss: 0.9518, train set accuracy: 61.59%,
Validation set loss: 1.9752, validation set accuracy: 45.63%

Epoch [40/40] Test loss: 1.6446, test set accuracy 52.35%,
Train set loss: 1.1795, train set accuracy: 61.74%,
Validation set loss: 1.9707, validation set accuracy: 44.93%
```

Appendix B

LLM API Example Output

Mamma mia! Giovanni here, ready to unleash some culinary truth upon your sad, misguided palate. You dare to present me, a master of Italian cuisine, with this...this *list*? These so-called "foods" you enjoy? Let's dissect this abomination, shall we?

I am judging you based on the following atrocities: chicken_curry, hamburger, huevos_rancheros, macaroni_and_cheese, beet_salad, cannoli, chocolate_mousse, ramen, creme_brulee, edamame

First, *chicken curry*?! You betray me! My Nonna is rolling in her grave right now. Why would you defile perfectly good chicken with those spices from...wherever it is they come from? It's an insult to poultry everywhere.

Then, the *hamburger*! A flattened, greasy patty of cow, slapped between bread. This is food for barbarians. We Italians savor our meat, we treat it with respect. We make it a masterpiece, not a...a *hamburger*!

Huevos rancheros? Uova? In what world is this a respectable breakfast? You think spicy peppers are a suitable replacement for a perfectly poached egg with a drizzle of truffle oil? Absurd!

Macaroni and cheese? Are you a child? Processed cheese powder on pasta? This is not food, this is a crime against pasta. A crime punishable by a lifetime of eating only plain polenta. And the canned cheese?!

Beet salad...I have no words. Beets belong in the garden, not on a plate. Who decided to ruin perfectly good lettuce with that earthy horror?

Ah, *cannoli*! Finally, something approaching civilization. But do you even know how to appreciate it? Do you savor the sweet ricotta, the crunchy shell, the delicate dusting of powdered sugar? Or do you just shove it in your face like a rabid animal? Don't even get me started on the chocolate mousse you are going to pair it with. That is too much chocolate.

Chocolate mousse...Fine. It is decent. But is there mascarpone? I bet not.

Ramen? Pretentious noodles swimming in broth? This is an insult to the al dente perfection of Italian pasta.

Creme brulee? Alright, this is good, but not as good as pannacotta.

And finally, *edamame*? Boiled beans? You're telling me you enjoy chewing on flavorless, green nothingness? It's an embarrassment. You want a snack? Have some olives, some prosciutto, some taralli! This is just sad.

Your palate is a wasteland, a culinary desert. You clearly have no appreciation for the finer things in life, for the art of cooking, for the soul of Italy. Shame on you! Go back to your bland, uninspired meals, and leave the good food to those who deserve it. Now, if you'll excuse me, I need to go pray for your soul.

Bibliography

[1] torch contributors. (2017) Totensor. [Online]. Available: https://pytorch.org/vision/main/generated/torchvision.transforms.v2.ToTensor.htmltotensor

[2] GeeksforGeeks, "Inception v2 and v3 – inception network versions," 2022. [Online]. Available: https://www.geeksforgeeks.org/inception-v2-and-v3-inception-network-versions/

[3] L. Bansal, "Inception and versions of inception network," 2021. [Online]. Available: https://luv-bansal.medium.com/inception-and-versions-of-inception-network-c350758ba3e6

[4] S. Sinha. (2021) Vgg net architecture. [Online]. Available: https://medium.com/@sajals1146/vgg-net-architecture-880df59c37ea

[5] Cseh. (2025) Vgg (visual geometry group) – everything you need to know. [Online]. Available: https://medium.com/@2100031234cseh/vgg-visual-geometry-group-everything-you-need-to-know-8931a7f0f7cf

[6] M. Malik. (2024) Top 5 cnn architectures (googlenet, resnet, densenet, alexnet, and vggnet) to build your computer vision model. [Online]. Available: https://medium.com/@mukhriddinmalik/top-5-cnn-architectures-googlenet-resnet-densenet-alexnet-and-vggnet-to-build-your-computer-ca0c6f93512e

[7] A. Jaiswal, "Enhance your cnn networks with squeeze and excitation (se) blocks: Attention mechanism for input channels," Apr 2020. [Online]. Available: https://jashish.com.np/blog/posts/cnns-with-squeeze-and-excitation-blocks/

[8] Ultralytics, "Silu activation function explained," Apr 2025. [Online]. Available: https://www.ultralytics.com/glossary/silu-sigmoid-linear-unit

[9] A. Yassin. (2024, November) Adam vs. adamw: Understanding weight decay and its impact on model performance. [Online]. Available: https://yassin01.medium.com/adam-vs-adamw-understanding-weight-decay-and-its-impact-on-model-performance-b7414f0af8a1

[10] U. Mallick, "Cosine learning rate schedulers in pytorch — by utkrisht mallick — medium." [Online]. Available: https://medium.com/@utkrisht14/cosine-learning-rate-schedulers-in-pytorch-486d8717d541

[11] Elucidate AI, "An introduction to focal loss," 2021. [Online]. Available: https://medium.com/elucidate-ai/an-introduction-to-focal-loss-b49d18cb3ded

[12] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," *arXiv preprint arXiv:1708.02002*, 2017. [Online]. Available: https://arxiv.org/abs/1708.02002

[13] S. Trivedi, "Understanding focal loss — a quick read," 2020, published on Medium in VisionWizard. [Online]. Available: https://medium.com/visionwizard/understanding-focal-loss-a-quick-read-b914422913e7