# Assignment: Food Classification with CNN

Aleksandra Bobrova (s3660141), Ryan Dorland (s3219992), Sam Chaman Zibai(s1678876), Ksenia Shileeva(s3971449)

## 1 INTRODUCTION

Food recommendation systems have the potential to revolutionize the way people discover and enjoy food. This project explores the implementation of a deep learning-based food classification engine using Convolutional Neural Networks (CNNs), designed to infer user preferences.

Convolutional Neural Networks (CNNs) are a type of neural network where layers use convolutions, which apply small filters (sets of weights) across different parts of the input. While traditional neural networks learn large weight matrices, CNNs reuse a small set of weights, making them more efficient and particularly effective at detecting local patterns in data such as images.

This project involves designing a CNN that works as a food recommendation system. By analyzing food images uploaded by users, the system classifies dishes into one of 91 categories and uses their results to build a personalized taste profile. The core task centers around building a robust CNN model from scratch in PyTorch to accurately perform this classification task on a substantial real-world dataset.

## 2 DATASET

The dataset used in this project is divided into two primary sets: a training set and a test set. The training set contains approximately 45k images, while the test set includes around 22k images. Each image is labeled as belonging to one of 91 distinct food categories. These categories range from commonly known dishes like pizza and sushi to more niche items such as specific regional meals.

Each class of food is represented by a directory containing its respective images, which makes it compatible with PyTorch's `ImageFolder` dataset loader. One of the key components of machine learning is splitting the data into a training and validation set, this is so that we can test out our trained model on new and unseen data. For our dataset we split the original dataset into a training and validation set with an $85:15$ ratio. For datasets larger than ten thousand a $80:20$ or $90:10$ split is common, we opted for the mean of these two ratios.

A fixed random seed ensures that this split is consistent across runs, which is critical for fair comparison and reproducibility in deep learning experiments.

## 3 DATA PREPROCESSING AND AUGMENTATION

Real-world images often contain a great deal of variability due to differences in lighting, perspective, camera quality, and background noise. To account for this, data augmentation is used to improve the generalization of the CNN model. The training dataset undergoes several transformations, including random horizontal flips (with a 50% probability), small random rotations (up to $\pm 15$ degrees), resizing to a fixed input dimension of $256 \times 256$ pixels, and normalization using the standard ImageNet means and standard deviations.

Normalization plays an important role in ensuring that the input data exhibits zero mean and unit variance across each channel, which facilitates faster convergence and enhanced stability in the learning progress. On the other hand, Image augmentation acts as a form of regularization, forcing the network to learn invariance to small perturbations. This increases the model's ability to generalize outside of the training data, especially when dealing with diverse real-world, user-uploaded images.

For the validation and test sets, only resizing and normalization are applied. This avoids introducing artificial distortions that could bias the evaluation, together with having reliable simulation of the inference phase during training.

## 4 MODEL ARCHITECTURE

The model is a custom-built CNN inspired by architectural elements found in state-of-the-art models such as ResNet [1]. CNNs are well-suited for image classification tasks due to their ability to learn spatial hierarchies of features, starting from low-level edges to high-level semantic concepts.

### 4.1 Convolutional Base

The network begins with a convolutional block consisting of a $7 \times 7$ convolution layer with stride 1 and padding 3, followed by batch normalization and a ReLU activation. This layer is designed to quickly capture coarse-level spatial patterns such as contours and texture regions, and is followed by a $3 \times 3$ max pooling operation that reduces the spatial resolution while keeping key features.

The motivation behind using batch normalization is to standardize the input to each layer, which reduces internal co-variate shift and allows for faster and more stable training. Moreover, it eliminates the necessity to use bias neurons in the preceding convolutional layers, which explains the choice of using `bias=False` in our implementation. ReLU, or in some cases LeakyReLU, introduces non-linearity while avoiding the vanishing gradient problem common with traditional sigmoid or tanh activations. LeakyReLU is used after a dense layer and every convolution layer, except the first one, since it produced better performance for our architecture and the dataset. This activation function prevents some neurons from dying if their weights were pushed to the negative range during training.

### 4.2 Residual Learning

The core of the model consists of multiple residual blocks. Each residual block contains two convolutional layers, with batch normalization and activation functions, along with a shortcut connection that bypasses the convolutional layers. This design enables each block to learn the residual $F(x) = H(x) - x$ between the identity function $x$ and the desired output $H(x)$. By learning residuals instead of directly modeling the transformations, the model mitigates the degradation problem observed in very deep networks.

Given the relatively smaller size of our architecture compared to ResNet-50, we implemented a basic type of a residual block instead of the BottleNeck type [2]. These blocks are stacked progressively, increasing the number of feature channels while reducing spatial dimensions using strided convolutions.

The architecture flows as follows:

- Initial convolution and max pooling
- Two blocks with 64 channels
- Two blocks with 128 channels
- Two blocks with 256 channels
- Two blocks with 512 channels
- Dense layer with 8192 inputs and 1024 outputs
- Dense layer for final classification with 91 outputs

This hierarchy looks at increasingly abstract visual features across layers and is reflective of how CNNs are structured in architectures like ResNet-18.

### 4.3 Classifier Head

After the final convolutional stage, the feature maps are pooled using average pooling, reducing each feature map to a $4 \times 4$ spatial size. The output is then flattened and passed through a fully connected layer consisting of 1024 hidden units. A LeakyReLU activation is applied to introduce non-linearity to the model, followed by dropout with a rate of 0.6. Dropout works as a regularizer by randomly deactivating neurons during training, which reduces the risk of overfitting.

Finally, a linear output layer maps the 1024-dimensional hidden representation to the 91 output classes. This layer computes the logits (raw scores) that are later used together with a softmax function (which is internally done by the `CrossEntropyLoss function`) to determine class probabilities.

## 5 TRAINING STRATEGY

The network is trained from scratch using the Stochastic Gradient Descent (SGD) optimizer with a learning rate of 0.001, momentum of 0.8, and a weight decay of $10^{-4}$. SGD with momentum allows the optimizer to build velocity in the direction of minima, which helps escape shallow local minima and avoid oscillations. The weight decay serves as L2 regularization, which is a technique used to reduce the risk of overfitting, penalizing very large weights and promoting generalization.

The `CrossEntropyLoss` function is used as the objective, which is well-suited for multi-class classification tasks. This function combines a softmax activation with the negative log likelihood loss, encouraging the model to assign high probability to the correct class while minimizing the probabilities of incorrect classes. The training process is executed over 70 epochs with a batch size of 32. Accuracy and loss are monitored for both training and validation phases at each epoch. These metrics will also be used later to construct a plot of the training progress. From the resulting graph, we can identify the optimal balance point between bias and variance for the model and load the weights corresponding to that point. Additionally, the model with the best validation accuracy is saved, but it might not be used in the final version, where the decision is based on the progress graph.

To decide on the best model architecture and fine-tune the hyperparameters, all the models were trained on the LIACS computers, with their training progress on the validation set being analyzed later. This allowed for the final choice of the model, located in the `Assignment.ipynb` file. Notably, most of the models during the tuning phase were not run for the maximum number of epochs, because their worse performance was noticed on earlier epochs.

## 6 PERFORMANCE EVALUATION

### 6.1 Training and Validation Performance

Throughout training, we logged and saved intermediate training statistics including epoch-wise accuracy and loss for both the training and validation sets. These metrics were visualized using progress plots to help diagnose learning behavior and identify the optimal stopping point. The plots in Figure 1 present training versus validation accuracy (left) and training versus validation loss (right) across 70 epochs.

As expected in deep neural network training, both training loss and training accuracy improved steadily throughout the epochs. However, the validation accuracy began to flatten around epoch 35-40 and showed minor improvements afterward. While training accuracy continued to increase above 85%, validation accuracy peaked and started to fluctuate, indicating that the model is starting to overfit. Moreover, the validation loss started to increase after the same point of 35-40 epochs. This tells us that all the points after that will have poor generalization on the test set, again due to overfit.

To avoid overfitting while still achieving strong performance, we selected epoch 46 as the model checkpoint for evaluation on the test set. The accuracy and loss at epoch 46 and surrounding points are:

```
Epoch [44/70]:
    Train Loss: 0.5625 | Train Acc: 82.98%
    Val Loss: 1.9898 | Val Acc: 55.92%
Epoch [45/70]:
    Train Loss: 0.5284 | Train Acc: 84.12%
    Val Loss: 1.9988 | Val Acc: 56.65%
Epoch [46/70]:
    Train Loss: 0.5049 | Train Acc: 84.62%
    Val Loss: 1.9713 | Val Acc: 57.33%
Epoch [47/70]:
    Train Loss: 0.4730 | Train Acc: 85.66%
    Val Loss: 2.0205 | Val Acc: 57.77%
Epoch [48/70]:
    Train Loss: 0.4501 | Train Acc: 86.30%
    Val Loss: 2.1411 | Val Acc: 56.02%
```

This epoch represents a good trade-off between learning capacity and generalization, just before the validation performance began to get worse or oscillate.

### 6.2 Final Test Set Performance

We evaluated the saved model from epoch 46 on the separate test set of $\sim 22,000$ images. The performance was measured in terms of

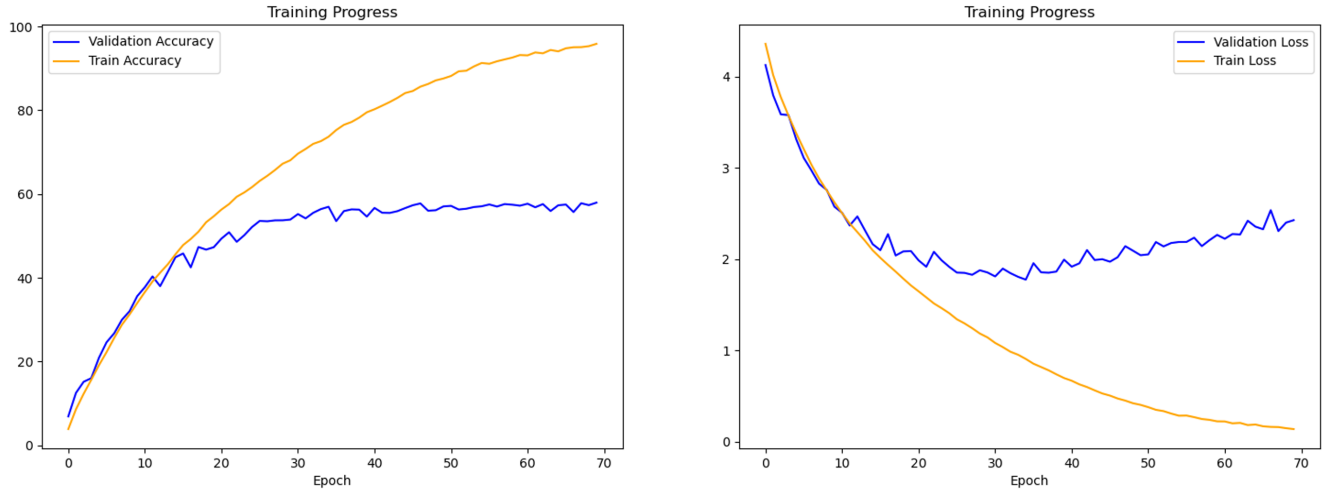


**Figure 1: The left plot shows accuracy, and the right shows loss for both training and validation. Validation accuracy peaks around epoch 46, after which overfitting begins, which can be seen from the continued rise in training accuracy and drop in training loss, while validation metrics do not improve anymore.**

both accuracy and cross-entropy loss, resulting in a test accuracy of 57.21 and a test loss of 1.96.

The accuracy on the test set is only 0.1% less than on the validation set, whereas the test loss showed even better results, being slightly less than validation one. This supports the model's ability to generalize to unseen data. The test accuracy reflects decently strong performance given the challenge of classifying among 91 different food categories, particularly with no pretrained weights.

## 7 SIMULATED PREFERENCE DETECTION

To simulate the user experience, we randomly sampled 10 images from the test set to mimic user uploads. These images were processed by the trained model, and their predicted food categories were recorded. The frequency of each predicted category across the 10 samples gives an idea of the user's taste profile.

This list can be used to create a user taste vector, which can inform for example restaurant recommendations by matching dish profiles with restaurant menus.

## 8 PROFILE OF PREFERENCE GENERATION

To enhance the interpretability of the system, we extended the preference detection system by integrating a large language model (LLM), to generate natural language summaries of the predicted preferences. The goal of this component was to convert a list of food classes after prediction into human readable taste profiles.

Our initial idea was utilizing Llama 3 model by adapting the tutorial from Langchain with Llama 3.2:1b website [3]. However, installing Llama model on the LIACS machines requires admin permissions. Thus, we used the model Mixtral-8x7B-Instruct-v0.1 via HuggingFace API[4].

The LLM was prompted to write a short paragraph describing the user's food preference based on the predicted list of classes. The prompt is fed and a short paragraph summarizing the user's preferences is requested. To prevent the language model's output from redundantly repeating the input prompt, the prompt text is excluded before presenting the final description to the user.

## 9 DISCUSSION/CONCLUSION

Our project shows an effective application of a custom-built CNN for multi-class food image classification. Reaching a test accuracy of 57.21 and a test loss of 1.96 means that our model was able to learn some meaningful representations, despite the challenge of classifying across 91 classes.

The strength of our implementation lies in the use of techniques such as residual learning, batch normalization, LeakyReLU activations and dropout to help reduce the chance of overfitting. The accuracy on the test only being 0.1% less than on the validation set shows that the model performs well and consistently on unseen data.

However the accuracy of 57.21 also shows the limitations of our model. The complexity of the dataset is high, which makes classification more difficult, especially if no pretrained models are used to find deeper semantic features.

To simulated a practical case, we implemented a user preference detection system based on random test samples. This shows the potential use of our model in a real world setting. We also integrated a LLM to translate these prediction into a natural language description for improved interpretability of the output.

For future improvements, we could look at transfer learning using pretrained models which could significantly boost our accuracy. Additionally, adding multimodal inputs forms such as text or metadata along with the images could enhance the accuracy and improve the models performance.

Overall, our model provides a solid foundation for food classification and taste profiling with a user oriented functionality.

## REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[2] Y. Shafiei, "Resnet-from-scratch," 2023. Accessed: 2025-05-14.

[3] S. Katiyar, "Langchain with llama 3 2.1b," 2024. Accessed: 2025-05-14.

[4] Mistral AI, "Mixtral-8x7b-instruct-v0.1," 2023. Accessed: 2025-05-16.