

Neural Computing Assignment: Food classification (CNN)

Simone de Vos Burchart (s1746995) & Hao Chen (s3990788)
Group Number: 36

1 INTRODUCTION

This assignment focuses on developing a deep learning-based food classification system using Convolutional Neural Networks (CNNs), under strict computational constraints. The main objective is not only to achieve high accuracy but also to explore how to design, train, and optimize a CNN model from scratch without any pre-trained weights.

2 LOADING THE DATASET

The dataset is provided in the form of two folders (train and test). Both folders contain 91 subdirectories which represent the classes, and each class contains approximately 500 images. We read the images from both folders using the cv2 library, and record their class labels. We studied the images to see what kind of pre-processing we would need to do.

We use the `image.shape()` function to know the majority of images have a height or width of 256 pixels. Initially, we tried resizing all images to 256×256 , but it required excessive GPU memory. Therefore, we resized the images to 64×64 , which still preserved enough detail for training while being memory-efficient. This resizing also ensures that all inputs to the CNN have a consistent size, which is required for batch processing.

We normalize the images so that their values are between 0 and 1, to make it easier for the CNN to use them.

Next we divide the entire train images set into `train_dataset` (90%) and `val_dataset` (10%) by using `sklearn's train_test_split()` function with a fixed random state (=1) to ensure reproducibility. Since we use a train dataset and a validation dataset, they serve different purposes. So we use different transformation pipelines to make the features of images more significant, which will help the model to learn the features of images. **Transformations applied to the training data:**

- `RandomHorizontalFlip(p=0.5)`: Simulates mirror views of food items to improve generalization.
- `RandomRotation(degrees=30)`: Introduces small-angle rotations to improve rotational robustness.
- `RandomResizedCrop(scale=(0.6, 1.0))`: Simulates zoom by randomly cropping part of the image.
- `ColorJitter(brightness=0.1, contrast=0.1)`: Applies slight changes in brightness and contrast to simulate lighting variations.
- `RandomGrayscale(p=0.1)`: Converts the image to grayscale with a 10% chance, to force the model to focus on texture instead of just color.
- `RandomAffine`: Simulates different camera angles.
- `RandomErasing(p=0.25)`: Erases parts of the images to encourage the model not to rely on specific areas.
- `Normalize(mean=[0.5]*3, std=[0.5]*3)`

These data augmentation techniques are crucial to prevent overfitting. Since the training dataset is relatively limited compared to the 91 food categories, applying random transformations helps the model see slightly different variations of each image in every epoch. This reduces the chance of memorizing the training data and forces the model to learn more robust, generalizable features. Some transformations are rather aggressive, but they gave us good results, even better than when using dropouts in our model.

Transformations applied to validation and test data:

- `Normalize(mean=[0.5]*3, std=[0.5]*3)`

The transformation we applied for validation and test data ensure that all input images have a consistent distribution, without any randomness. This guarantees that the evaluation of the model's performance are deterministic and fair.

For the loaders of all three datasets (`train_dataset`, `val_dataset` and `test_dataset`), we set the `batch_size` to 32 and `num_workers` to 0 despite the hardware configuration of the lab computers, which have 8 CPU cores. We wanted to make sure our code would be reproducible, so `num_workers=0` seemed the safest option. We chose a small batch size to make the model generalize better, and to keep the memory usage down.

In addition, we set `shuffle=True` and use a fixed generator = `g` (`seed 42`) for the training Dataloader, that ensures that the training data is randomly shuffled each epoch while maintaining reproducibility.

3 CNN MODEL

We implemented a custom CNN architecture inspired by ResNet, named FoodCNN. The model is composed of four main stages (Block1 to Block4), each consisting of two residual blocks. These blocks help address the vanishing gradient problem by introducing identity skip connections that allow the gradient to propagate more effectively during training.

Each ResidualBlock includes two convolutional layers with batch normalization and ReLU activations. The purpose of the ResidualBlock is to reduce the spatial dimensions of the feature maps while increasing their semantic richness. We are using the convolutional layer with a stride of 2 to halve the width and height of the feature maps. Using stride-2 convolutions allows the model to efficiently compress spatial information, reduce computational load, and focus on higher-level abstract features, and it is essential for deep image classification tasks.

As the network progresses, the number of feature channels increases, enabling the model to capture richer semantic representations. We finish with a fully connected layer (classifier) that includes an adaptive average pooling layer and flatten layer that condense

the spatial dimensions. Finally, we have a linear layer that produces a class score for each class.

The FoodCNN model's forward function simply passes the input through the four blocks and the classifier.

4 TRAINING THE MODEL

To train our FoodCNN model from scratch, we implemented a full training loop that includes early stop, forward and backward propagation, optimizer updates, and accuracy tracking. To ensure reproducibility and consistency on the university machines, we fixed all relevant random seeds and enabled deterministic CUDA behavior (`torch.backends.cudnn.deterministic = True`). If we don't set this setting, the CUDA may choose non-deterministic convolution algorithms, resulting in slightly different gradients each time.

4.1 Optimizer and Learning Rate Strategy

We selected the Adam optimizer to help our CNN training. Adam (short for Adaptive Moment Estimation) combines the advantages of two other widely used optimizers: AdaGrad and RMSProp. It maintains per-parameter learning rates and uses momentum from both first and second moments of the gradients. Compared to standard SGD, the Adam optimizer enables faster and more stable convergence in deep neural networks. This is especially beneficial during the early stages of training, when gradients may be unstable or noisy due to random initialization or unbalanced data. A faster convergence allows the model to reach an acceptable performance level within fewer epochs, which is crucial under limited compute resources or training time constraints (which is suitable for our assignment setting, where model development is expected to be completed within a short period and under limited hardware conditions).

However, one disadvantage of Adam is that it is less sensitive to manual learning rate and may sometimes lead to suboptimal convergence or large weight magnitudes. To address this, we integrated the `torch.optim.lr_scheduler.ReduceLROnPlateau`. This scheduler monitors the validation accuracy and reduces the learning rate when improvements stop, helping the model fine-tune its parameters and avoid overshooting. We started with an initial learning rate of $1e-4$ and used `ReduceLROnPlateau` to automatically reduce the learning rate by a factor of 0.3 whenever the validation accuracy did not improve for 2 consecutive epochs. And the learning rate would keep decreasing but never go below a minimum threshold of $1e-7$, ensuring steady yet controlled convergence throughout training.

4.2 Early Stopping & Training Loop

We created an `early_stop` function that monitors the validation accuracy after each epoch. If the validation accuracy does not improve for 10 consecutive epochs, we stop the training process early. On one hand, this can prevent overfitting by halting training once the model stops generalizing better on the validation set. On the

other hand, it saves training time, especially since we set the total number of epochs to 100.

During each epoch, we created a `train()` function to set the model to training mode and iterate over mini-batches from the training set. For each batch, we first clear the previous gradients using `optimizer.zero_grad()` function and then perform a forward pass to compute predictions and loss (by using `criterion(outputs, labels)`). After that we call the `loss.backward()` function, which computes the gradients of the loss with respect to all learnable parameters in the model.

We create the `validate()` function to evaluate the model on the validation dataset without updating the weights of the model. Which means we set the model to evaluation mode with the `model.eval()` function. During validation, we compute the forward pass and track the average loss and accuracy across all batches.

For the visualization, we use `tqdm()` to display loss and accuracy for each batch during the training and validation. After each epoch, if the validation accuracy is higher than all previous records, we save the current model as '`best_model.pth`'. In addition, we save the full training state as a checkpoint so we can resume the training if we interrupted the training.

5 CALCULATING MODEL PERFORMANCE

After training, we evaluated our model on the unseen test set to measure its generalization capability. The evaluation procedure on the test set closely follows the same methodology used for validation. Based on the final test result, our model achieved a test accuracy of 61.06% and an average test loss of 1.57. This indicates that the model generalizes reasonably well to unseen data, and its performance on the test set is comparable to that on the validation set, suggesting stable learning.

6 SIMULATION OF RANDOM USER & USE AN LLM TO GENERATE A DESCRIPTION

After verifying that our model performs reasonably well, we explored how it could be applied in real world scenarios. One such application is food category recognition using our trained FoodCNN model. We randomly selected 10 images from the test dataset and used our model to classify them. If we compare the classification result with real labels, we find out 6 of 10 predictions were correct. This relatively low accuracy can be attributed to the limitations of our CNN model, which could be improved with further optimization or data augmentation.

Beyond classification, we also demonstrate how we can help LLMs by using our model. After classifying the user's uploaded images into food categories, we can use an LLM to interpret the result. The LLM summarizes the user's possible food preferences based on the predicted labels from our model.

7 CONCLUSION

In this project, we developed a custom CNN model for food classification from scratch. Through careful pre-processing, data augmentation, and trying many different training strategies, we achieved a final test accuracy of 61.06%. This indicates that our model generalizes reasonably well, though there is room for improvement, especially in a classification task involving 91 categories.

During development, we encountered frequent overfitting – the model tended to memorize training set details while failing to generalize to unseen data. To mitigate this, we reduced the learning rate when validation performance plateaued, and enhanced visual diversity through image-level augmentations. These strategies effectively reduced overfitting and improved the model’s robustness. A

future improvement could come from adding dropouts to the model, however for the locations and dropout rates we tried, our model did not perform as well as without dropout. With a systematic hyperparameter optimization method like Bayesian Optimization, the optimal dropout locations and rates, as well as the other hyperparameters, can be determined.

For another future improvement, we could explore training deeper ResNet variants (e.g., ResNet50), leveraging pretrained weights via transfer learning, or employing semi-supervised learning to incorporate unlabeled data. These approaches could significantly boost generalization performance while reducing the need for large, manually labeled datasets.