7 Transfer Learning and Fine-Tuning

Chapter Abstract. This chapter builds a Food-101 image classifier with fast.ai, then shows the same fine-tuning pattern in a smaller Keras 3 example. The main move is simple: keep an ImageNet-pretrained convolutional body, replace the ImageNet head with a Food-101 head, train the new head first, and then fine-tune the body cautiously. That fast.ai workflow introduces source and target tasks, frozen and unfrozen stages, splitters, discriminative learning rates, default data transforms, validation discipline, reproducibility records, and common debugging failures. A completed ResNet34 validation run supplies the baseline numbers, and the official Food-101 test split is reserved for the selected final model.

7.1 Why Transfer Learning Is an Applied Deep Learning Shortcut

Transfer learning is an applied deep learning shortcut because it spends target-task training budget adapting reusable source-task features instead of relearning every visual pattern from scratch. Training a classifier for Food-101, a benchmark dataset of food photographs from 101 dish categories, from random initialization would spend early epochs relearning image features that a good pretrained model may already have: edges, color boundaries, textures, and object parts. Transfer learning starts from source-task weights and uses the target-task budget to adapt those features instead of rebuilding the whole representation from random initialization [206, 47].

Food-101 keeps the shortcut concrete. The dataset contains food photographs from 101 dish categories, so the classification goal is easy to picture, but the visual problem is still demanding. A classifier must separate close classes such as ravioli, dumplings, and gnocchi while handling different camera angles, plates, lighting, backgrounds, and serving styles. Starting from a natural-image model lets the experiment spend its limited training budget on food-specific adaptation.

Pause and predict

Before reading the transfer-learning setup, predict which part of an ImageNet classifier should transfer most directly to Food-101 and which part should be replaced. Name two visual cues that might transfer, then name one reason the ImageNet classifier head is a poor final layer for Food-101.

The source model in this chapter is a ResNet-family classifier pretrained on ImageNet, a large-scale natural-image classification benchmark. ImageNet and Food-101 do not ask for the same labels, so pretraining is a starting bet,

not evidence of accuracy. The bet is reasonable because both datasets contain natural photographs, and food photographs reuse many cues learned from broader image collections: color boundaries, specular highlights, object contours, repeated textures, and arrangements of parts [42, 239, 91, 310]. The ImageNet head still predicts ImageNet classes, not Food-101 dishes, so the workflow keeps the reusable body, replaces the head, and fine-tunes the resulting model on Food-101.

Replace the head. Transfer reuses visual features, not the old source-task label vocabulary.

Applied work should report accuracy and runtime together. A model that gains a small amount of validation accuracy after tripling training time may be a poor choice for a first baseline, while a slightly smaller model that trains quickly and is reproducible may be more valuable. For this chapter, the first useful result is not a leaderboard score. It is a strong, auditable baseline: a compact fast.ai script, an explicit validation split, a reserved test split, recorded versions and hardware, and a training recipe whose mechanics can be explained rather than treated as magic [105].

7.2 A CNN Before the Classifier as a Feature Extractor

A convolutional classifier can be separated into two conceptual parts. Let x be an RGB Food-101 image. A feature extractor, also called a body in fast.ai's vision helpers, maps the image to a representation r:

$$g_{\theta_{\text{body}}}(x) = r.$$

The classifier head maps that representation to class logits:

$$h_{\theta_{\text{head}}}(r) \in \mathbb{R}^{101}.$$

The complete Food-101 classifier is the composition

$$f_{\theta_{\text{body}},\theta_{\text{head}}}(x) = h_{\theta_{\text{head}}}(g_{\theta_{\text{body}}}(x)).$$

Read Figure 7.1 from left to right: pixels become the learned representation r through the pretrained body, and the new head turns r into Food-101 logits.

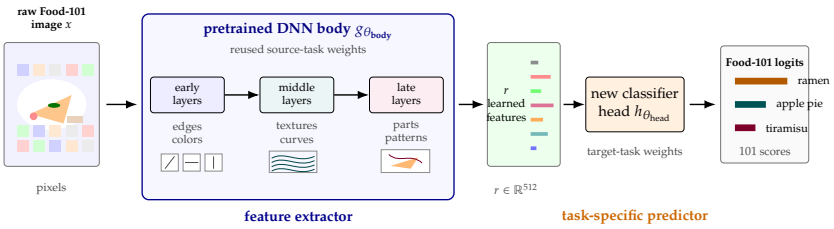

Figure 7.1: A DNN body used as a feature extractor. The body transforms raw pixels into a learned representation r that records useful visual evidence; the classifier head is the smaller target-specific part that converts those features into Food-101 logits. Transfer learning keeps or cautiously fine-tunes the body while replacing the source-task head.

Here θ_{body} denotes the body parameters and θ_{head} denotes the head parameters. The logits are 101 real numbers because Food-101 has 101 target categories; the largest softmax probability usually determines the top-1 prediction.

Definition 7.2.1 (Feature extractor and classifier head) *A feature extractor is the part of a model that converts raw input into learned features useful for prediction. A classifier head is the task-specific part that maps those features to logits for the target classes. In a transfer-learning classifier, the feature extractor usually starts from source-task weights, while the target-task head is newly created or reinitialized.*

The Food-101 intuition follows directly from the equation. Early and middle convolutional layers can respond to edges, color contrasts, textures, curves, and object parts. Bread crust texture, leafy salad shapes, sauce color, grill marks, plate boundaries, and spoon-like contours are not identical to ImageNet objects, but they are visual patterns in the same broad family of natural photographs. The final ImageNet layer is different. It may contain a logit for "golden retriever" or "espresso maker," but Food-101 needs logits for "apple pie," "ramen," and "tiramisu." That mismatch explains why transfer learning normally replaces the head even when it keeps most of the body.



Figure 7.2: Synthetic close-up food panels showing visual cues that can be useful before the final classifier head: crust texture, grill marks, sauce gloss, leafy curves, plate boundaries, and repeated noodle texture. These cues are not Food-101 labels by themselves; they are examples of reusable visual evidence that a pretrained convolutional body may already represent.

A small dense example makes the cost difference visible.

Pause and predict

Suppose a pretrained body outputs a 512-dimensional feature vector and the new Food-101 head maps it to 101 logits. Before reading the arithmetic, compute the number of weights and biases in a single linear head. Which part of the model should be cheaper to train first: the new head or the full ResNet body?

Consider a two-layer model

$$f(x) = W_2\phi(W_1x + b_1) + b_2.$$

If $r = \phi(W_1 x + b_1)$ is treated as the learned feature vector, then replacing only W_2 and b_2 is cheaper than learning both layers from scratch. Suppose a pretrained body produces a 512-dimensional vector and the target task has 101 classes. A single linear Food-101 output layer has

$$512 \times 101 + 101 = 51{,}813$$

trainable parameters. A ResNet body has millions of convolutional parameters, so a frozen-body first stage can train a comparatively small target-specific layer before risking changes to the pretrained representation.

Cheap first stage. Training the new head first updates tens of thousands of parameters before touching millions in the body.

Definition 7.2.2 (Transferable representation) *A transferable representation is a representation learned on one task that improves target-task learning compared with starting from random weights under the same training budget and evaluation protocol.*

Calling a representation transferable is an empirical claim. It counts for Food-101 only when it improves the training and validation outcome under a controlled comparison.

7.3 Source Task, Target Task, Transfer Learning, and Fine-Tuning

Precise vocabulary prevents several common mistakes. The source dataset and source task are the data and prediction problem used before target training begins. In this chapter, the source task is ImageNet classification, where a ResNet learned to map natural images to ImageNet categories [42, 239, 91]. The target dataset and target task are the new data and prediction problem. Here the target task is Food-101 classification, where the model must map food photographs to 101 dish labels [15].

Definition 7.3.1 (Pretraining and transfer learning) *Pretraining is training on a source task before target-task training starts. Transfer learning is the use of source-trained parameters, source-trained representations, or both, for a target task.*

In notation, a source task draws examples from one distribution $\mathcal{D}_{\text{source}}$, while the target task draws examples from $\mathcal{D}_{\text{target}}$. Pretraining finds parameters that work well for the source objective,

$$\theta_{\text{pre}} \approx \arg\min_{\theta} \mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{source}}}[\ell(f_\theta(x), y)].$$

Transfer learning starts the target problem from θ_{pre} instead of from random weights. Domain shift means $\mathcal{D}_{\text{source}}$ and $\mathcal{D}_{\text{target}}$ are not the same, so validation decides how much of the source representation is useful.

Validation decides transfer. Similar source data makes transfer plausible, but target validation is the evidence.

Definition 7.3.2 (Frozen parameter and fine-tuning) *A frozen parameter is a parameter whose value is not updated by gradient descent during a training stage. Fine-tuning is continued training from pretrained weights on the target task, usually after replacing or reinitializing the target-specific head.*

The mathematical definition is clean, but fast.ai has one important training-policy wrinkle. Section 7.6 notes that batch-normalization layers can remain trainable under the learner's `train_bn` policy even when earlier splitter groups are otherwise frozen.

The two terms are related but not interchangeable. Transfer learning names the overall strategy: use something learned from the source task when solving the target task. Fine-tuning names the target-task training procedure that adapts the transferred model. In the Food-101 workflow, `vision_learner` constructs the transfer-learning model, while `learn.fine_tune(...)` runs a particular fine-tuning schedule for that model.

Freezing is a training protocol rather than a change in the mathematical model. During a frozen stage, the forward pass still uses the pretrained convolutional body to compute $g_{\theta_{\text{body}}}(x)$, but the optimizer does not update most of θ_{body}. The new head parameters θ_{head} do update, so the model learns to map pretrained visual features to Food-101 logits. During unfrozen fine-tuning, the optimizer is allowed to update all or more of θ_{body} as well as θ_{head}.

The two stages differ in which parameters are allowed to minimize the target risk. A frozen-head-training stage has the form

$$\min_{\theta_{\text{head}}} \frac{1}{n} \sum_{i=1}^{n} \ell\left(h_{\theta_{\text{head}}}(g_{\theta^{\text{pre}}_{\text{body}}}(x_i)), y_i\right),$$

while an unfrozen stage also updates some or all of θ_{body}. The model function may look the same in the forward pass; the optimizer's parameter set is what changes.

Transfer learning can fail. If the source and target domains are too different, the source features may be poorly matched to the target. If the learning rate is too high after unfreezing, useful pretrained features can be damaged. If validation data leaks into model selection or the official test split is checked repeatedly, reported accuracy becomes unreliable. The check that matters is validation performance under a fixed split, followed by one reserved final test evaluation after model selection.

7.4 Food-101: Data, Split, Metric, and Baseline Goal

Food-101 contains 101 food categories and 101,000 images. The standard split provides 750 training images and 250 test images per class, giving

$$101 \times 750 = 75{,}750$$

training-split images and

$$101 \times 250 = 25{,}250$$

official test images [15]. The workflow here carves a validation set from the official training split and reserves the official test split for a final evaluation. With a 20% validation holdout from the training split, the approximate counts are 60,600 training images and 15,150 validation images before exact splitter rounding.

Figure 7.3 uses open-license substitute images with selected Food-101 class names. These are not examples from the Food-101 dataset, but they show the range of the target domain: familiar categories such as apple pie, pizza, tacos, sushi, and ramen alongside visually close classes such as dumplings, ravioli, and gnocchi. That mixture explains why transfer learning is useful but still needs validation evidence.

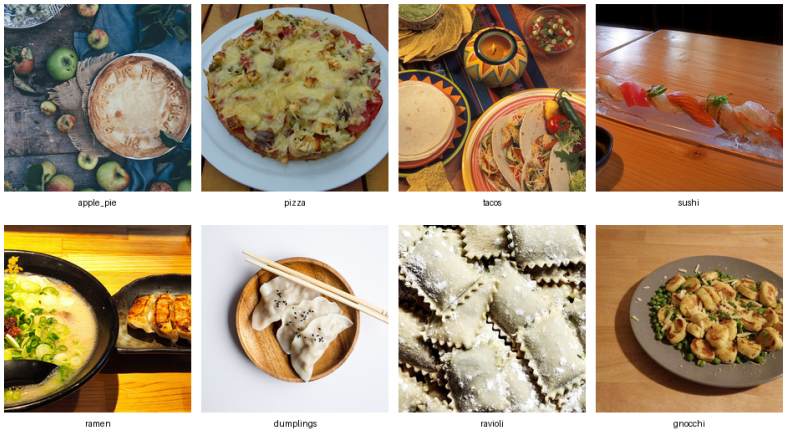

Figure 7.3: Open-license Food-101-like images cropped and resized to 224 × 224 pixels, matching the chapter's training image size. They illustrate selected Food-101 class labels without reproducing Food-101 dataset images. Source and license details are recorded with the figure assets; the selected sources are CC0 or public domain.

Image credit note. Source and license details are recorded with the figure assets. The selected source images are CC0 or public domain, and no share-alike or attribution-required image is used in this figure.

The primary metric is top-1 accuracy, the fraction of examples for which the highest-scoring class is correct. Top-5 accuracy can also be informative for Food-101 because several categories are visually similar, but the first baseline should report top-1 accuracy, runtime, hardware, and enough configuration information to reproduce the run. A responsible report records the data source, split rule, model architecture, pretrained weights, image size, batch size, training schedule, learning rate, device, seed, validation accuracy, final test accuracy, elapsed time, and artifact paths. Section 7.12 returns to this checklist after the code and fine-tuning mechanics are visible, then records the measured ResNet34 baseline.

7.5 Minimal fast.ai Food-101 Fine-Tuning Code

A minimal fast.ai baseline should show the whole path from files to a trained learner without hiding the validation decision. The following code uses the Food-101 split files distributed with the dataset. It trains and validates only on entries from `train.txt`; `test.txt` is loaded separately so it can remain untouched until final evaluation.

Test split discipline. Keep the official test list out of tuning decisions; use it once after model selection.

```
from fastai.vision.all import *
from torchvision.models import resnet34, ResNet34_Weights

path = untar_data(URLs.FOOD)

def food_items(split_name):
    txt = path / f"{split_name}.txt"
    if not txt.exists():
        txt = path / "meta" / f"{split_name}.txt"
    return L(txt.read_text().splitlines()).map(
        lambda item: path / "images" / f"{item}.jpg"
    )

train_valid_files = food_items("train")
test_files = food_items("test")  # Reserve for one final evaluation.

food = DataBlock(
    blocks=(ImageBlock, CategoryBlock),
    get_y=parent_label,
    splitter=RandomSplitter(valid_pct=0.2, seed=42),
    item_tfms=Resize(460),
    batch_tfms=aug_transforms(size=224),
)

dls = food.dataloaders(train_valid_files, bs=64)
dls.show_batch(max_n=9)

learn = vision_learner(dls, resnet34, weights=ResNet34_Weights.DEFAULT,
                       metrics=accuracy)
learn.fine_tune(5, base_lr=3e-3)
```

Pause and predict

Before reading the code explanation, inspect the `DataBlock`. Predict where each image label comes from, which file list is used for training and validation, which split remains reserved, and what batch tensor shape should result from `bs=64` and 224-by-224 RGB crops.

The call to `untar_data(URLs.FOOD)` downloads and expands Food-101 in fast.ai's data directory. The helper `food_items` reads entries such as `apple_pie/1005649` from a split file and converts each entry to an image path under `images`. The split files are usually in the dataset root; the small fallback handles package layouts that keep them under a `meta` directory. Because the parent directory of `images/apple_pie/1005649.jpg` is `apple_pie`, the label function `parent_label` recovers the class name directly from the path.

The DataBlock states the data contract. The input block is an image, the target block is a category, the splitter makes a reproducible 20% validation set from train_valid_files, and the transforms describe how raw images become batches. Resize(460) first makes individual images large enough for random crops, while aug_transforms(size=224) creates 224-by-224 transformed training batches. Section 7.11 unpacks the fast.ai default augmentation policy used by this baseline. With batch size 64, the resulting PyTorch tensor has shape (64, 3, 224, 224) in batch, channel, height, width order.

The learner line is the transfer-learning setup. vision_learner receives the data loaders, a ResNet34 architecture, pretrained TorchVision weights, and an accuracy metric. It constructs a model with 101 output logits because the data loaders know the Food-101 vocabulary. Unlike the CIFAR-10 ResNet examples in Chapter 6, this workflow keeps the ImageNet-style pretrained stem: Food-101 photographs are resized to ImageNet-scale 224-by-224 crops, and changing the stem would discard the source weights that transfer learning is trying to reuse. The final line is the fine-tuning call: it asks fast.ai to adapt that transferred model for five unfrozen epochs after its default frozen warm-up stage. The companion script in code/chapter_transfer_learning_fastai/food101_fastai.py expands this small example into a command-line run that records metadata, parameter counts, and smoke-test evidence. The optional notebook code/chapter_transfer_learning_fastai/food101_fastai_walkthrough.ipynb follows the same route interactively, while keeping the script as the reproducible source of truth.

7.6 What vision_learner Builds Under the Hood

The learner helper vision_learner turns the data loaders and architecture into a transferred classifier. The data loaders supply batches, labels, transforms, and the number of classes. The helper loads the requested architecture with pretrained weights when weights are provided, removes the source-task classification end, attaches a new head, installs a splitter for parameter groups, and freezes the pretrained groups by default for transfer learning [58]. When normalize=True and the model is pretrained, fast.ai also adds an image normalization transform using the statistics associated with the pretrained weights, unless the data loaders already contain one.

Source trace: fast.ai vision learner docs describe create_body, create_head, pretrained normalization, splitters, and default transfer-learning freezing.

Definition 7.6.1 (Splitter and parameter group) *A splitter is a function that maps a model to ordered parameter groups. A parameter group is a collection of parameters trained with shared optimizer hyperparameters, such as a learning rate. Freezing, progressive unfreezing, and discriminative learning rates act on these groups.*

For a TorchVision ResNet, fast.ai's model metadata specifies where the architecture should be cut to form the body. The body keeps the pretrained convolutional feature extractor, including the ImageNet stem, and drops the original ImageNet average-pooling and fully connected classifier. That is the opposite of the CIFAR-10 adaptation issue from Chapter 6: there,

small 32×32 images motivate replacing the ImageNet stem; here, Food-101 images are resized to 224-by-224 crops, so preserving the pretrained stem is part of preserving the transferred representation. The new fast.ai head begins with adaptive pooling. With the common `concat_pool=True` setting, `AdaptiveConcatPool2d` concatenates adaptive average-pooling and adaptive max-pooling outputs, so a 512-channel feature map produces $2 \times 512 = 1024$ pooled features before flattening. Batch normalization, dropout, linear layers, and ReLU activations then form the trainable head, ending in a linear layer with 101 logits for Food-101.

Component	Example shape or role	Why it exists
Input batch	`(64, 3, 224, 224)`	64 RGB images in PyTorch order
ResNet body	feature maps with many channels	reusable visual representation
Adaptive concat pool	$2 \times 512 = 1024$ values	fixed-size vector from spatial maps
Hidden head layer	e.g. 512 hidden units	task-specific nonlinear remapping
Final head layer	$512 \times 101 + 101 = 51{,}813$ parameters	101 Food-101 logits

A splitter tells fast.ai which parameters should be trained together. It returns parameter groups, such as an early body group and a head group. Freezing operates on those groups: `freeze()` leaves the last group trainable and freezes earlier groups, while `unfreeze()` makes all groups trainable. One version-dependent caveat is batch normalization, whose train/eval behavior was introduced in Chapter 5. fast.ai's `Learner` has a `train_bn` setting that defaults to training frozen normalization layers during transfer learning, so "frozen" means frozen according to the splitter plus this normalization-layer exception [54]. Discriminative learning rates also operate group by group, which is how the same learner can support staged fine-tuning without exposing all of the optimizer plumbing.

7.7 What `fine_tune` Does: Two Training Stages

`Learner.fine_tune` is the adaptation procedure inside the transfer-learning workflow, not a synonym for transfer learning itself. It is also not a single undifferentiated training call. By default it first freezes the pretrained groups and trains the new head for `freeze_epochs`; then it unfreezes the model and trains for `epochs` using discriminative learning rates [53]. Both stages use `fit_one_cycle` unless the call is customized. A one-cycle learning-rate policy raises the learning rate and then anneals it during a short training run, connecting fast.ai fine-tuning practice to the schedule family introduced in Chapter 5 [261].

Source trace: current fast.ai source freezes, runs one-cycle, halves `base_lr`, unfreezes, then runs one-cycle with a discriminative learning-rate slice.

The frozen stage stabilizes the new head. The Food-101 head begins with random or newly initialized weights, so its first job is to learn a rough mapping from pretrained visual features to food classes. Updating only the

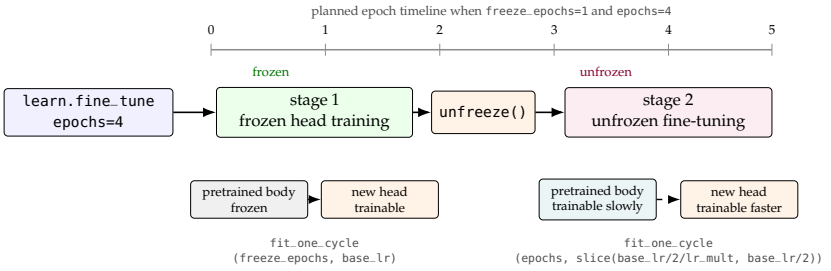

Figure 7.4: Two-stage mechanics of `Learner.fine_tune`. The first stage trains the newly attached Food-101 head while pretrained groups remain frozen. The second stage unfreezes the model and uses a lower discriminative learning-rate range so the body can adapt more cautiously than the head.

head is fast because most of the convolutional body is not being optimized. It also reduces the chance that a poorly initialized head and a large gradient step will immediately distort useful pretrained filters.

Pause and predict

Suppose `fine_tune` uses `freeze_epochs=1`, `epochs=4`, `base_lr=0.003`, and `lr_mult=100`. Before reading the calculation, predict how many epochs are frozen and unfrozen, then estimate the high and low ends of the unfrozen learning-rate range.

The unfrozen stage adapts the whole model. Once the head has learned a reasonable target-task mapping, the body can change slowly to better represent Food-101 textures and shapes. Early layers usually receive smaller learning rates than later layers and the head. If `freeze_epochs=1` and `epochs=4`, the planned training has one frozen epoch followed by four unfrozen epochs. In current fast.ai, the unfrozen stage first halves the supplied `base_lr`. If the call uses `base_lr=0.003` and keeps `lr_mult=100`, the high end of the unfrozen learning-rate range is

$$0.003/2 = 0.0015,$$

and the low end is

$$0.0015/100 = 0.000015.$$

The exact numbers matter because a factor-of-two difference in learning rate can change whether unfreezing improves validation loss or damages useful pretrained features.

Treat this two-stage schedule as a default to test. If validation loss jumps immediately after unfreezing, the unfrozen learning rate may be too high, the validation split may be unstable, or the target data may require a gentler schedule. Validation curves decide whether the second stage helped.

7.8 Same Fine-Tuning Concepts in Keras 3

The fine-tuning pattern should survive a change of library. The Keras 3 version below writes the steps out directly: load an ImageNet-pretrained body, replace the classification head, train the new head while the body is frozen, then unfreeze the body and continue with a much smaller learning rate.

The fast.ai code remains the measured Food-101 baseline for this chapter, but Keras makes it easier to see which pieces are library-independent. Keras 3 is a useful translation target because the backend is selected before `import keras`, and the model code can run on supported TensorFlow, PyTorch, or JAX backends when the installed backend stack supports the chosen layers and data path [134, 136]. The cells below are the notebook route. They keep the same workflow visible in smaller pieces: setup first, pretrained body second, Food-101 head third, frozen-head training fourth, and unfrozen fine-tuning last.

Notebook cell 1: select the backend and import libraries

```
import os
os.environ.setdefault("KERAS_BACKEND", "tensorflow")

import keras
from keras import layers
from keras.applications import ResNet50
from keras.applications.resnet import preprocess_input

NUM_CLASSES = 101
IMAGE_SIZE = 224
```

The backend selection must happen before Keras is imported. The constants keep the target-task vocabulary size and image size visible in the run record. The snippets below assume that `train_ds` and `val_ds` have already been built from the Food-101 training split as Keras-compatible datasets yielding `(image_batch, label_batch)` pairs; the split rule should match the fast.ai run if the two libraries are being compared.

Notebook cell 2: load the ImageNet-pretrained body

```
base_model = ResNet50(
    include_top=False,
    weights="imagenet",
    input_shape=(IMAGE_SIZE, IMAGE_SIZE, 3),
    pooling="avg",
)
base_model.trainable = False
```

The Keras ResNet application keeps the ImageNet-pretrained feature extractor and removes the original 1,000-class head [139]. Freezing the base model prepares the first stage so that only the newly attached Food-101 head can learn.

Notebook cell 3: attach augmentation and a Food-101 head

```
inputs = keras.Input(shape=(IMAGE_SIZE, IMAGE_SIZE, 3), name="image")
x = layers.RandomFlip("horizontal")(inputs)
x = layers.RandomRotation(0.05)(x)
x = layers.RandomZoom(0.10)(x)
x = preprocess_input(x)
x = base_model(x, training=False)
x = layers.Dropout(0.2)(x)
outputs = layers.Dense(NUM_CLASSES, name="food101_logits")(x)
model = keras.Model(inputs, outputs)
```

The new `Dense(NUM_CLASSES)` layer supplies Food-101 logits. The augmentation and preprocessing are part of the model path here, so the tensor that reaches `base_model` has the format expected by the pretrained ResNet.

Notebook cell 4: train the new head while the body is frozen

```
loss = keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=3e-4),
    loss=loss,
    metrics=["accuracy"],
)
model.fit(train_ds, validation_data=val_ds, epochs=1)
```

While `base_model.trainable = False`, this frozen stage trains the replacement head against the Food-101 labels without changing the transferred convolutional body.

Notebook cell 5: unfreeze and fine-tune cautiously

```
base_model.trainable = True
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=1e-5),
    loss=loss,
    metrics=["accuracy"],
)
model.fit(train_ds, validation_data=val_ds, epochs=5)
```

The important freeze/unfreeze behavior is in the `trainable` flags and the second `compile`. After `base_model.trainable = True`, recompiling with a lower learning rate gives Keras a new training configuration in which trainable body weights can change cautiously [140]. Because the model graph above calls `base_model(x, training=False)`, BatchNorm-style training behavior and running-statistic updates stay in inference mode even during this cautious fine-tuning stage; changing that call is a separate, documented experiment rather than an implicit part of the translation.

Concept	fast.ai expression	Keras 3 expression
Pretrained body	`vision_learner(..., weights=...)`	`ResNet50(..., include_top=False, weights="imagenet")`
New head	built by `vision_learner` from the data vocabulary	explicit `Dense(101)` logits layer
Frozen stage	`learn.freeze()` or the first stage of `fine_tune`	`base_model.trainable = False`, then `compile` and `fit`
Unfrozen stage	`learn.unfreeze()` plus `fit_one_cycle`	`base_model.trainable = True`, then recompile with a lower learning rate
Discriminative learning rates	`slice(low, high)` over splitter groups	custom training logic or staged partial unfreezing; the compact example uses one conservative low rate

Keras exposes more moving parts, while fast.ai supplies more transfer-learning policy. In Keras, the base model, head, `trainable` flags, losses, optimizers, and recompilation are visible in ordinary model code. What it does not provide here is fast.ai's one-line splitter-based discriminative learning-rate behavior. To reproduce that behavior exactly in Keras, the experiment would need a custom training loop, optimizer logic with different learning rates for different variable groups, or a staged policy that unfreezes selected layers in separate runs. For this chapter, the important lesson is the invariant structure: transfer learning replaces the source-task head, and fine-tuning controls which parts of the transferred model are allowed to move.

The companion script `code/chapter_transfer_learning_fastai/food101_keras3.py` expands this snippet into a runnable check with backend selection, synthetic smoke mode, Food-101 split-file loading, and the same reserved-test-set discipline used by the fast.ai script.

7.9 Fine-Tuning More Layers: Splits and Progressive Unfreezing

The default two-stage recipe trains the head, then the whole model. Sometimes Food-101 benefits from a middle step: unfreeze the later ResNet blocks first, watch whether validation loss improves, and only then train the full body. This approach is often called progressive unfreezing. Each stage increases the number of trainable parameters, so the validation curve and runtime record must earn the extra cost.

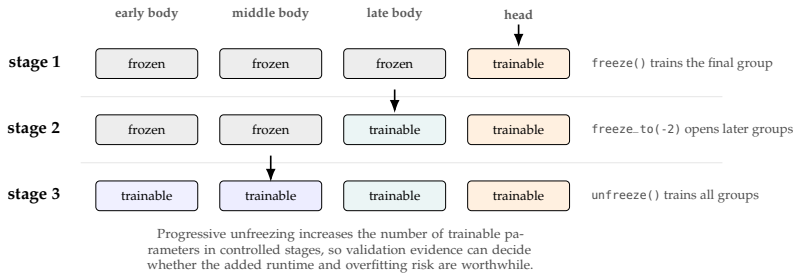

Figure 7.5: Progressive unfreezing as an ordered parameter-group experiment. Each row trains more of the pretrained body than the previous row, moving from a cheap head-only adaptation toward full-model fine-tuning.

A staged fast.ai recipe can be written directly once the learner has the splitter introduced in Section 7.6. With a built-in fast.ai splitter for a TorchVision model, the groups are chosen from the architecture metadata and the learner can use `freeze()`, `freeze_to`, and `unfreeze()` without writing a custom splitting function:

```
learn.freeze()
learn.fit_one_cycle(1, 3e-3)

learn.freeze_to(-2)
learn.fit_one_cycle(2, slice(1e-5, 1e-4))

learn.unfreeze()
learn.fit_one_cycle(3, slice(1e-6, 1e-4))
```

The first stage trains only the final group, typically the head. The second stage, `freeze_to(-2)`, trains the last two groups while earlier groups stay frozen. The third stage trains all groups. The learning-rate slices are deliberately small after the head stage because pretrained body features should usually change more gently than a newly initialized head.

Unfreeze gradually. Each stage buys more adaptability at the cost of more overfitting risk and runtime.

If the built-in split is too coarse for the question being tested, a custom ResNet splitter can separate major body blocks:

```
from fastai.vision.all import L, params

def resnet_block_splitter(m):
    return L(m[0][:4], m[0][4], m[0][5], m[0][6], m[0][7], m[1:]).map(params
    )
```

The exact module indices depend on the model object created by the current fast.ai and TorchVision versions, so the script should print or inspect the model before relying on a custom splitter. More trainable layers can improve adaptation to Food-101, but they also increase runtime and overfitting risk. Extra unfreezing earns its place only if the validation gain is large enough to justify the added stage.

7.10 Different Learning Rates for Different Blocks

Discriminative learning rates use different learning rates for different parameter groups. The intuition is asymmetrical. Early convolutional filters often represent general image patterns such as edges and color contrasts, so they should change slowly. Later blocks and the new head are closer to the target labels, so they may need larger updates. This idea is common in practical fine-tuning recipes and is built into fast.ai's slice-based learning-rate interface [105, 54, 53].

Definition 7.10.1 (Discriminative learning rate) *A discriminative learning rate scheme assigns different learning rates to different parameter groups in the same model, usually with smaller rates for earlier pretrained groups and larger rates for later groups or the new head.*

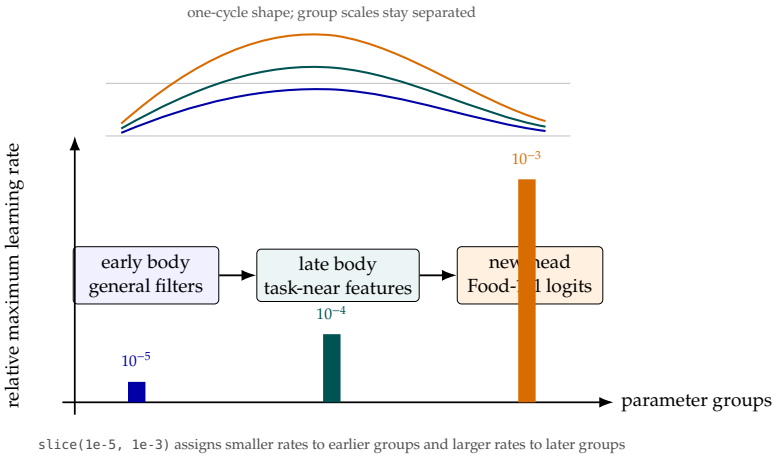

Figure 7.6: Discriminative learning rates across parameter groups. A one-cycle schedule may raise and lower each group's rate during an epoch cycle, but the head still receives larger updates than early pretrained filters when the learning-rate range spans several orders of magnitude.

When a fast.ai fit call receives `slice(low, high)`, it assigns a range of learning rates across the learner's parameter groups. With three groups and `slice(1e-5, 1e-3)`, the group learning rates can be summarized as follows.

Group	Example role	Approximate maximum learning rate
1	early body	1×10^{-5}
2	late body	1×10^{-4}
3	head	1×10^{-3}

The one-cycle schedule changes learning rates during training, but the groups keep their relative scale. A simple update-size calculation shows why this matters. If a parameter has gradient magnitude 0.2, a learning rate of 10^{-5} gives an update magnitude of $0.2 \times 10^{-5} = 0.000002$, while a learning rate of 10^{-3} gives $0.2 \times 10^{-3} = 0.0002$. The latter is 100 times larger. For Food-101 fine-tuning, that difference lets the head adapt quickly while early pretrained filters move cautiously.

7.11 Default fast.ai Batch Transforms

Earlier chapters used augmentation mainly as a simple regularization idea: show the model slightly different views of the same labeled image during training. The Food-101 fast.ai baseline deserves a more concrete reading because one short line,

```
batch_tfms=aug_transforms(size=224)
```

expands into a whole default image policy. It is still a baseline default, not a result to optimize blindly, but it is useful to know what the default is asking the data loader to do.

There are two related preprocessing stages in the minimal code. First, `item_tfms=Resize(460)` works on individual PIL images before they become tensor batches. With the default crop-style resize, training examples can be cropped from different positions, while validation examples use a deterministic center crop on the cropped dimension [57]. This creates a large enough intermediate image for the next stage. Second, `batch_tfms=aug_transforms(size=224)` works on mini-batches after tensor conversion, usually on the accelerator. It samples the final 224-by-224 training view and applies the default fast.ai geometric and lighting perturbations.

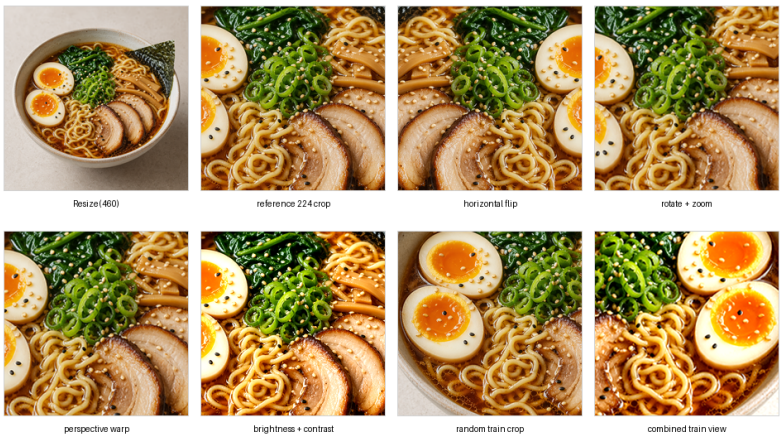

Figure 7.7: Deterministic visual guide to the fast.ai default augmentation families used by the Food-101 baseline. The source is an AI-generated food photograph created for this textbook, not a Food-101 dataset image. The flip, rotation/zoom, warp, and lighting panels use the same 224-by-224 reference crop so their visual effects can be compared directly; the random-crop and combined-view panels illustrate separate training-style samples. Actual training samples random values, so a real mini-batch will not match these fixed panels exactly.

Table 7.1 translates the most important defaults into modeling language. The line does not mean that every training image receives every transformation. It defines a stochastic policy: some images are flipped, some receive affine or lighting changes, and the sampled magnitudes vary from batch to batch [57].

The label-preservation rule is the reason these defaults are conservative. A transform is safe only when it normally keeps the class label true. Left-right flipping is usually safe for Food-101, but vertical flipping is not a good default. Mild crop, zoom, rotation, warp, brightness, and contrast changes are usually safe because they imitate normal photographic variation. Stronger crops can remove the food, strong rotations can create implausible scenes, and aggressive color changes can damage useful cues such as browning, sauce color, or cooked texture.

Augmentation rule. A transform is regularization only if the label should stay true after the transform.

The important experimental point is where the policy lives. Randomized image transforms belong in the training data loader. Validation and final test examples should be processed deterministically so they measure performance on a stable input distribution. fast.ai's random transforms are training-split transforms by default, and the deterministic validation path is part of why the same `DataBlock` can support both training and evaluation [57]. Chapter 5 returns to augmentation as a broader training and regularization topic; here, keep `aug_transforms(size=224)` fixed for the first Food-101 baseline and change it only in a controlled comparison against the same validation split.

Table 7.1: Main defaults behind `aug_transforms(size=224)` for this chapter's fast.ai baseline.

Transform family	Default setting	Practical meaning for Food-101
Horizontal flip	`do_flip=True`; `flip_vert=False`; flip probability 0.5	A pizza, salad, or bowl of ramen usually keeps the same label after a left-right flip. The default avoids vertical flips, which would create unnatural upside-down plates and serving contexts.
Affine geometry	`p_affine=0.75`; `max_rotate=10`; `min_zoom=1.0`; `max_zoom=1.1`	Small rotations, translations induced by cropping, and mild zoom make the classifier less dependent on exact camera framing. The limits are deliberately modest so the dish remains visible.
Perspective warp	`max_warp=0.2`; applied through the affine/coordinate transform path	A small warp simulates camera viewpoint changes. It is useful for real food photographs, but a large warp could distort shape cues that separate similar classes.
Lighting	`p_lighting=0.75`; `max_lighting=0.2` for brightness and contrast	Food photos vary by exposure, shadows, and restaurant lighting. Brightness and contrast perturbations help the model avoid treating a lighting condition as a class feature.
Output and padding	`size=224`; `mode='bilinear'`; `pad_mode='reflection'`	The transformed batch matches the ResNet input scale. Reflection padding avoids introducing black borders when rotations, shifts, or warps sample outside the original image boundary.
Strength knob	`mult=1.0`	Increasing `mult` strengthens rotation, warp, and lighting together; decreasing it weakens them together. Treat this as an experiment knob, not a hidden default to change without validation evidence.

7.12 Practical Recipe for High Accuracy and Fast Training

A good Food-101 workflow begins with a fast pretrained ResNet34 or ResNet50 baseline, not with an uncontrolled search over many settings. Use the provided training split, reserve the official test split, carve out a validation split, and record runtime next to accuracy. fast.ai defaults are a reasonable first baseline because they combine pretrained normalization, resizing, augmentation, one-cycle scheduling, frozen head training, and unfrozen fine-tuning in a compact API [58, 53]. Use them as starting defaults, then test changes against the same validation split.

Controlled comparisons should change one major factor at a time. A model comparison might hold image size, batch size, split, learning rate, and epoch count fixed while changing ResNet34 to ResNet50. An image-size comparison should keep the model and schedule fixed. A progressive-unfreezing comparison should keep the architecture fixed and vary only the freeze schedule. Mixed precision is often useful when supported by the device and numerically stable, but it should also be recorded because it changes runtime and memory behavior.

A related fast.ai trick is progressive resizing: start with a smaller training image size so early epochs are cheaper, then rebuild the data pipeline at the final size and continue fine-tuning from the same weights [105]. In Food-101 terms, a controlled experiment might begin with smaller crops such as 160×160 or 192×192, then continue at the chapter's default 224×224 size while keeping the validation split fixed and reducing batch size if memory becomes tight. The possible gain is that the model first learns coarse structure quickly and only later pays the cost of higher-resolution detail. The risk is that the extra stage adds runtime without producing a meaningful validation improvement. Treat progressive resizing as an experiment knob, not a free accuracy coupon.

One practical bridge between a fixed default and an uncontrolled learning-rate search is fast.ai's learning-rate finder. After the learner is constructed, `learn.lr_find()` runs a short mock training pass in which the learning rate grows exponentially, records the training loss, stops early if the loss diverges, and restores the model afterward. The curve in Figure 7.8 shows the diagnostic pattern: very small rates make little progress, a middle range lowers the loss quickly, and very large rates can send the loss upward.

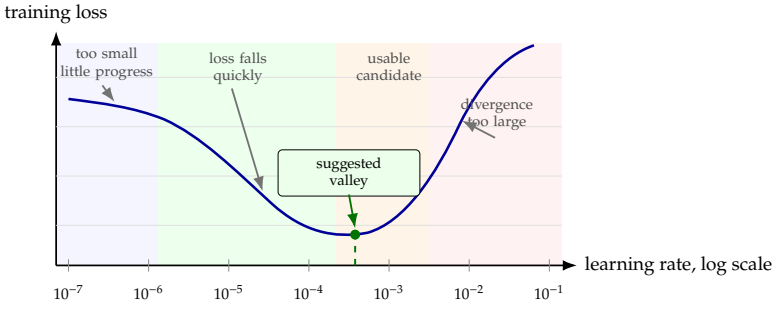

Figure 7.8: Conceptual fast.ai learning-rate finder curve. `learn.lr_find()` increases the learning rate during a short mock training pass and records how training loss responds. The valley suggestion is a candidate maximum learning rate for a follow-up run; validation evidence still decides whether that choice is useful.

The returned suggestions, such as the default `valley` suggestion, are candidates for the maximum learning rate used by `fit_one_cycle` or `fine_tune`; they are not proof that the setting is best [260, 53]. A follow-up Food-101 experiment could record the suggestion and then validate the resulting `base_lr` against the same split:

```
suggested_lr = learn.lr_find().valley
learn.fine_tune(5, base_lr=suggested_lr)
```

Record field	Recorded Food-101 validation baseline
Run type	Full Food-101 baseline, not a smoke-test proxy
fast.ai / timm version	fast.ai 2.8.7; timm not installed
PyTorch / TorchVision	PyTorch 2.11.0; TorchVision 0.26.0; Pillow 12.2.0
Data path and split rule	`fastai.URLs.FOOD`; `RandomSplitter(valid_pct=0.2, seed=42)` over the official training split produced 60,600 training items and 15,150 validation items
Hardware and device	AArch64 Linux runtime with fast.ai selecting a CUDA device; the compact checked-in artifact did not capture the host name
Architecture and weights	ResNet34 with pretrained TorchVision weights
Parameter counts	21,863,744 total parameters; approximately 596,096 trainable parameters in the frozen-head stage; all 21,863,744 parameters trainable after unfreezing
Batch size and image size	Batch size 64; resize size 460; training image size 224
Schedule and learning rate	One frozen-head epoch followed by five unfrozen epochs; `base_lr=3e-3`; unfrozen discriminative learning-rate ratio `lr_mult=100`
Frozen-stage outcome	492.917s training, 77.627s validation, validation loss 1.8694, validation accuracy 52.73%
Unfrozen-stage outcome	2594.293s training, 77.703s validation, validation loss 0.8861, validation accuracy 77.30%
Final test protocol	The 25,250 official test images remain reserved for `-evaluate-test` after model selection
Validation artifact record	The companion code includes the compact `reference_artifacts/food101-fastai/food101-baseline.csv` summary, plus command, version, metadata, and metric records in the same directory. The aggregate baseline fields used by the chapter are printed in this table. The full run also wrote `stage_metrics.csv`, `training_log.csv`, `result_summary.csv`, validation error-analysis CSVs, and plot files in its output run directory, but those larger raw artifacts are not committed

These records came from the companion script on 2026-04-22:

```
python food101_fastai.py \
  --architecture resnet34 --image-size 224 --resize-size 460 \
  --batch-size 64 --freeze-epochs 1 --epochs 5 --base-lr 3e-3
```

The mirrored metrics record reports 3,872.965 seconds of total elapsed script time, including data preparation, training, validation, and artifact writing. The official test split is reserved for the selected final model, and the validation accuracy is a measured Food-101 validation baseline.

Recorded check	Model and data	Schedule	Metric record	Runtime and artifacts
Dep. check	Food-101 mode configured for ResNet34, pretrained weights, image size 224, resize size 460, batch size 64	No data loader or training; `-check-deps` exits after import checks	Software and detected-device versions recorded	Writes compact version, metadata, and metric records in the configured output run directory
Synthetic smoke	Tiny generated image set, small architecture, no pretrained download unless explicitly requested	One short code-path check	Local validation and runtime values written by the smoke command, not bundled as a reference Food-101 result	Use only to verify the software path; do not cite as dataset evidence

Read the evidence through the cost column. The dependency and smoke checks verify that the software path works, while the full baseline gives one real Food-101 validation result for the default ResNet34 recipe. Do not promote a slower ResNet50 run or a progressive-unfreezing run just because it is more elaborate: the validation gain has to be large enough to justify the runtime and the noise of a single split.

The baseline artifacts also define the first error-analysis pass. After reproducing the run, inspect `validation_top_confusions.csv` for concentrated mistakes among visually close dish families such as dumplings, ravioli, and gnocchi, then inspect `validation_top_losses.csv` or `validation_top_losses.png` to separate plausible ambiguous photos from preprocessing or label problems. A top-loss image that contains several dishes or a cropped plate edge is often an ambiguous validation example; an image whose filename, class folder, or visual content clearly disagrees with the target is a data-block or label-audit issue; a repeated pattern of tiny food regions, heavy borders, or center-cropped ingredients points back to resize, crop, or augmentation settings. The aggregate baseline summary is printed above, but the raw run directory is the place where image-level validation evidence belongs. The high-accuracy path is incremental: establish a reproducible

baseline, inspect validation curves and errors, make one controlled change, record the result, and evaluate the official test split once after selecting the model from validation evidence.

7.13 Common Failure Modes and Debugging

The general training checklist from Chapter 5 still applies, but transfer learning has a shorter list of first suspects: labels from paths, frozen versus unfrozen parameter groups, pretrained normalization behavior, and validation transforms. Many transfer-learning failures are bookkeeping failures before they are modeling failures. If labels are wrong because paths were parsed incorrectly, even a strong pretrained model will learn the wrong mapping. Start debugging by showing a batch with labels and inspecting several file paths. For Food-101, an image under `images/apple_pie/...jpg` should receive the label `apple_pie`. If the label text is shifted, truncated, or taken from the wrong directory level, the data block should be fixed before training continues.

Validation mistakes are equally damaging. Random augmentation should not be applied as if validation were another training batch. The official test split should not be used repeatedly to choose learning rates, image sizes, or architectures. A clean workflow uses validation for model development and evaluates the reserved test split once after selection. If validation accuracy is surprisingly high, check for leakage; if validation accuracy is unstable, check the split size, random seed, class balance, and whether repeated runs agree.

Pause and predict

Suppose validation loss jumps immediately after unfreezing while training loss keeps improving. Before reading the table, predict two likely causes and one controlled response that changes only the fine-tuning schedule.

Symptom	Likely cause	Controlled response
Wrong labels in displayed batches	path parsing or label function error	inspect paths, labels, and class vocabulary
Validation loss jumps after unfreezing	learning rate too high or too many layers unfrozen	lower `base_lr`, use smaller slices, or unfreeze progressively
Out-of-memory error	batch size or image size too large	reduce batch size, reduce image size, or enable mixed precision when supported
Fast training but poor validation accuracy	head-only stage underfits or data transforms are wrong	check augmentation, train longer, inspect curves
Good validation, weak final test	validation overfitting or repeated test use	reset split discipline and report final test only once
Accuracy reported without context	missing reproducibility record	save versions, seed, command, device, runtime, and metrics

The strongest debugging habit is to change one thing at a time. Lowering the learning rate, changing the architecture, increasing image size, and changing augmentation in the same run may produce a better number, but it does not explain which choice mattered. Transfer learning is useful because it can be fast; controlled debugging preserves that advantage.

7.14 Summary

Transfer learning begins from the decomposition

$$f_{\theta_{\text{body}},\theta_{\text{head}}}(x) = h_{\theta_{\text{head}}}(g_{\theta_{\text{body}}}(x)).$$

The body g turns an image into learned features, and the head h maps those features to target-task logits. In Food-101, an ImageNet-pretrained ResNet body can provide useful natural-image features, while the ImageNet classifier head must be replaced by a Food-101 head with 101 outputs. This replacement is the central engineering move behind the short fast.ai learner construction line.

fast.ai packages the practical recipe into `vision_learner` and `fine_tune`. `vision_learner` sets up transfer learning: it creates a body and head, infers the output count from the data, applies pretrained normalization, installs a splitter, and begins with pretrained groups frozen. `fine_tune` then performs fine-tuning: it first trains the new head with the body frozen, then unfreezes and adapts the full model with discriminative learning rates. Keras 3 expresses the same structure through an explicit pretrained application model, an explicit dense head, `trainable` flags, and recompilation after trainability changes. Progressive unfreezing extends the same idea by exposing more groups and training them in a controlled

sequence. After the baseline is stable, stronger regularization and sizing comparisons such as MixUp, CutMix, label smoothing, and progressive resizing are reasonable next experiments.

The records prevent two common bad results: an accuracy number no one can reproduce, and a model tuned against the test set. Augmentation choices, the validation split, runtime, hardware, seeds, versions, and final test-set restraint belong beside the accuracy number. A pretrained body is not a free accuracy coupon; it is a starting bet whose value has to show up on the target validation split. A strong Food-101 classifier is valuable only when its accuracy, cost, and provenance are recorded well enough for another run to check it.

The body representation learned by the pretrained model becomes the feature-vector language of Chapter 8. The same source-task and target-task vocabulary also returns when Chapter 10 fine-tunes pretrained Transformer encoders and Chapter 11 studies pretrained instruction models.

7.15 Exercises

Answer the following ten questions in complete sentences unless a calculation is requested. When code appears, explain the purpose of the object or call rather than only naming it.

Exercise 1. Define source task and target task using ImageNet and Food-101 as the example.

Exercise 2. Write the formula $f(x) = h(g(x))$ and identify which part is the feature extractor and which part is the classifier head.

Exercise 3. Explain why the original ImageNet final layer cannot be used unchanged for Food-101.

Exercise 4. A pretrained body outputs a feature vector with 1,024 numbers. A new Food-101 linear classifier maps it to 101 logits. Compute the number of weights and biases in this final linear layer.

Exercise 5. Explain the difference between freezing the body and unfreezing the body.

Exercise 6. In fast.ai, why is it useful to train the new head before updating the whole pretrained model?

Exercise 7. A learner has three parameter groups: early body, late body, and head. If the groups use learning rates 10^{-5}, 10^{-4}, and 10^{-3}, which group changes fastest and why?

Exercise 8. Explain why a horizontal flip is usually safe for cat and food classification, but might be unsafe for digit recognition or text recognition.

Exercise 9. Read the Food-101 code in Section 7.5 and the Keras translation in Section 7.8. Match the data setup, pretrained body, new head, frozen stage, and unfrozen stage across the two implementations.

Exercise 10. Suppose validation accuracy improves after the frozen stage but gets worse immediately after unfreezing. Name two possible causes and two controlled fixes.

7.16 Homework: Train a Food-101 Classifier with fast.ai

Train a Food-101 image classifier using fast.ai and a pretrained ResNet-family model. Submit runnable code or a notebook plus a short report focused on practical evidence: model choice, data split, image size, batch size, learning rate, freeze schedule, validation accuracy, final test accuracy, runtime, hardware, and a brief error analysis. The report should explain transfer learning as the overall strategy and fine-tuning as the target-task adaptation procedure, using the language of this chapter: source task, target task, pretrained body, new head, frozen stage, unfrozen stage, and discriminative learning rates. The report fields exist because pretraining is a starting bet, not a free accuracy coupon: the bet only earns credit if validation evidence improves under a fixed split, cost record, and provenance.

Required Work

The minimum experiment uses the Food-101 dataset, states exactly how training, validation, and final test data are separated, and builds a learner with `vision_learner` and a pretrained ResNet-family architecture. It must include at least one standard `fine_tune` run and at least one controlled comparison, such as a different learning rate, architecture, image size, augmentation strength, or progressive-unfreezing schedule. Report validation accuracy and runtime for each run. Evaluate the selected model on the reserved test split once, and include a confusion matrix, top confusions, or a grid of high-loss examples. Do not tune repeatedly on the test set. Selected-model evidence: ____.