# ECE 60146 — Homework 9
# Transformers on Translation, CIFAR-10, and Food-101

Austin Toro

Spring 2026

## Contents

# 1 Task 1: Seq2seq Translation Warm-up (15 points)

## 1.1 (a) Code: Script used to run TransformerFG

```
class InstrumentedTransformerFG(TransformerFG):
    """TransformerFG wrapper that records metrics needed for the report."""

    def train_and_record(
        self,
        master_encoder: torch.nn.Module,
        master_decoder: torch.nn.Module,
        checkpoint_dir: str | Path,
    ) -> dict[str, Any]:
        checkpoint_dir = ensure_dir(checkpoint_dir)
        master_encoder.to(self.dl_studio.device)
        master_decoder.to(self.dl_studio.device)
        embeddings_generator_en = self.EmbeddingsGenerator(self, "en", self.embedding_size).to(self.dl_studio.device)
        embeddings_generator_es = self.EmbeddingsGenerator(self, "es", self.embedding_size).to(self.dl_studio.device)
        optimizer_params = self.optimizer_params
        beta1 = optimizer_params["beta1"]
        beta2 = optimizer_params["beta2"]
        epsilon = optimizer_params["epsilon"]
        master_encoder_optimizer = self.ScheduledOptim(
            optim.Adam(master_encoder.parameters(), betas=(beta1, beta2), eps=epsilon),
            lr_mul=2,
            d_model=self.embedding_size,
            n_warmup_steps=self.num_warmup_steps,
        )
        master_decoder_optimizer = self.ScheduledOptim(
            optim.Adam(master_decoder.parameters(), betas=(beta1, beta2), eps=epsilon),
            lr_mul=2,
            d_model=self.embedding_size,
            n_warmup_steps=self.num_warmup_steps,
        )
        criterion = nn.NLLLoss()
        num_sentence_pairs = len(self.training_corpus)
        batch_size = self.dl_studio.batch_size
        max_iters = math.ceil(num_sentence_pairs / batch_size)
        history: dict[str, Any] = {
            "loss_points": [],
            "learning_rate_points": [],
            "iterations_logged": [],
```

```
            "epoch_average_losses": [],
        }
        start_time = time.perf_counter()
        for epoch in range(self.dl_studio.epochs):
            random.shuffle(self.training_corpus)
            running_loss = 0.0
            epoch_running_loss = 0.0
            epoch_examples = 0
            for iteration in range(max_iters):
                left = iteration * batch_size
                right = min((iteration + 1) * batch_size,
                   num_sentence_pairs)
                batched_pairs = self.training_corpus[left:right]
                source_sentences = [pair[0] for pair in batched_pairs
                   ]
                target_sentences = [pair[1] for pair in batched_pairs
                   ]
                en_tensor = self.sentence_with_words_to_ints(
                   source_sentences, "en").to(self.dl_studio.device)
                es_tensor = self.sentence_with_words_to_ints(
                   target_sentences, "es").to(self.dl_studio.device)
                en_sentence_tensor = embeddings_generator_en(
                   en_tensor).float()
                es_sentence_tensor = embeddings_generator_es(
                   es_tensor).float()
                master_encoder_optimizer.zero_grad()
                master_decoder_optimizer.zero_grad()
                master_encoder_output = master_encoder(
                   en_sentence_tensor)
                predicted_word_logprobs, _ = master_decoder(
                   es_sentence_tensor, master_encoder_output)
                loss = torch.tensor(0.0, device=self.dl_studio.device
                   )
                for column in range(es_tensor.shape[1]):
                    loss = loss + criterion(predicted_word_logprobs
                       [:, column], es_tensor[:, column])
                loss.backward()
                master_encoder_optimizer.step_and_update_lr()
                master_decoder_optimizer.step_and_update_lr()

                normalized_loss = loss.item() / es_tensor.shape[0]
                running_loss += normalized_loss
                epoch_running_loss += normalized_loss * es_tensor.
                   shape[0]
                epoch_examples += es_tensor.shape[0]

                if iteration % 200 == 199:
```

```
                    avg_loss = running_loss / 200.0
                    running_loss = 0.0
                    current_lr = master_encoder_optimizer._optimizer.
                        param_groups[0]["lr"]
                    elapsed_seconds = time.perf_counter() -
                        start_time
                    print(
                        "[epoch:%2d/%d  iter:%4d  elapsed_time:%5d
                            secs] loss: %.4f  lr: %.8f"
                        % (
                            epoch + 1,
                            self.dl_studio.epochs,
                            iteration + 1,
                            elapsed_seconds,
                            avg_loss,
                            current_lr,
                        )
                    )
                    history["loss_points"].append(avg_loss)
                    history["learning_rate_points"].append(current_lr
                        )
                    history["iterations_logged"].append((epoch *
                        max_iters) + iteration + 1)

            epoch_average_loss = epoch_running_loss / max(
                epoch_examples, 1)
            history["epoch_average_losses"].append(epoch_average_loss
                )
            print(f"Finished epoch {epoch + 1}/{self.dl_studio.epochs
                } with mean loss {epoch_average_loss:.4f}")

        self.save_encoder(master_encoder)
        self.save_decoder(master_decoder)
        self.save_embeddings_generator_en(embeddings_generator_en)
        self.save_embeddings_generator_es(embeddings_generator_es)
        history["elapsed_seconds"] = time.perf_counter() - start_time
        return history
```

Listing 1: Instrumented TransformerFG training wrapper adapted from DLStudio's seq2seq example and training loop

## 1.2 (b) Figure: Training loss curve over 4 epochs

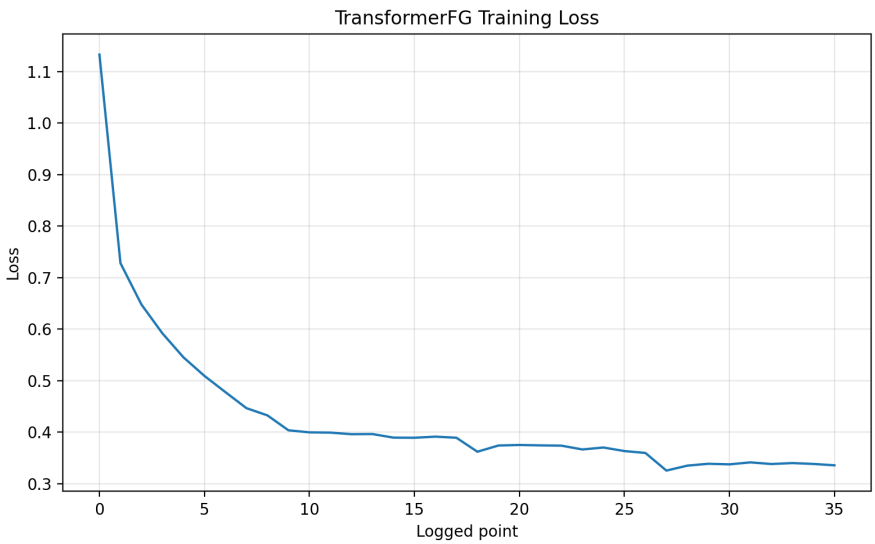



Figure 1: Task 1 TransformerFG training loss over the 4-epoch run. Logged once every 200 iterations.

## 1.3 (c) Text: 20 translations with correctness labels and accuracy fraction

Exact-match accuracy was 1/20. However, 15 additional outputs were at least partially semantically aligned with the target sentence, usually by preserving the main noun, verb, or sentence frame while dropping details or repeating words.

Table 1: Task 1 sampled translations with coarse correctness labels.

| # | English source | Spanish target | Model output | Label |
|---|---|---|---|---|
| 1 | SOS my down jacket keeps me quite warm EOS | SOS mi chaqueta de plumas me mantiene bastante calentito EOS | EOS mi me chaqueta chaqueta mantiene bastante bastante EOS EOS | semantically correct |
| 2 | SOS stop resisting EOS | SOS deja de resistirte EOS | EOS deja de EOS EOS EOS EOS EOS EOS EOS | semantically correct |
| 3 | SOS it is a beautiful day is not it EOS | SOS bonito día verdad EOS | EOS es un hermoso EOS EOS EOS EOS EOS EOS | wrong |
| 4 | SOS i never imagined meeting you here EOS | SOS nunca imaginé encontrarte aquí EOS | EOS nunca he reunión aquí aquí aquí EOS EOS EOS | semantically correct |
| 5 | SOS tom took mary home EOS | SOS tom llevó a mary a casa EOS | EOS tom se a a a a EOS EOS EOS | wrong |
| 6 | SOS i tried to get back to sleep EOS | SOS intenté volverme a dormir EOS | EOS intenté de a EOS EOS EOS EOS EOS EOS | semantically correct |

| # | English source | Spanish target | Model output | Label |
|---|---|---|---|---|
| 7 | SOS something remarkable happened EOS | SOS ocurrió algo increíble EOS | EOS algo algo EOS EOS EOS EOS EOS EOS EOS | wrong |
| 8 | SOS i am happy and satisfied EOS | SOS estoy feliz y satisfecho EOS | EOS estoy estoy satisfecho satisfecho EOS EOS EOS EOS EOS | semantically correct |
| 9 | SOS my apartment is near EOS | SOS mi apartamento está cerca EOS | EOS mi apartamento está cerca cerca EOS EOS EOS EOS | semantically correct |
| 10 | SOS we eat soup with a spoon EOS | SOS comemos la sopa con cuchara EOS | EOS comemos sopa con una EOS EOS EOS EOS EOS | semantically correct |
| 11 | SOS i do not have to answer that question EOS | SOS no tengo que contestar a esa pregunta EOS | EOS no tengo que que esa esa EOS EOS EOS | semantically correct |
| 12 | SOS this book is new EOS | SOS este libro es nuevo EOS | EOS este libro es nuevo EOS EOS EOS EOS EOS | correct |
| 13 | SOS you can talk to them EOS | SOS puedes hablar con ellos EOS | EOS puedes hablar con EOS EOS EOS EOS EOS EOS | semantically correct |
| 14 | SOS she is able to sing very well EOS | SOS ella puede cantar muy bien EOS | EOS ella es cantar muy muy muy EOS EOS EOS | semantically correct |
| 15 | SOS tom also has plans to go there EOS | SOS tom también planea ir allí EOS | EOS tom también tiene ir ir ir EOS EOS EOS | semantically correct |
| 16 | SOS even tom is surprised that mary lied EOS | SOS incluso a tom le sorprendió que maría mintiera EOS | EOS ni es sorprendió sorprendió sorprendió a mary EOS EOS | wrong |
| 17 | SOS i am in dire need of money EOS | SOS estoy mal de dinero EOS | EOS estoy en de de dinero EOS EOS EOS EOS | semantically correct |
| 18 | SOS this is the real world EOS | SOS este es el mundo real EOS | EOS éste es el el mundo EOS EOS EOS EOS | semantically correct |
| 19 | SOS i work in boston EOS | SOS trabajo en boston EOS | EOS trabajo en en EOS EOS EOS EOS EOS EOS | semantically correct |
| 20 | SOS he began to run EOS | SOS él empezó a correr EOS | EOS empezó empezó a EOS EOS EOS EOS EOS EOS | semantically correct |

## 1.4 (d) Text: Warm-up phase observations

The loss shows the expected warm-up dynamics. During the first 4,000 optimization steps the scheduled learning rate ramps from roughly $9.9 \times 10^{-5}$ to $1.98 \times 10^{-3}$, and the loss drops rapidly from about 1.13 at iteration 200 to about 0.37 near the end of warm-up. After step 4,000 the schedule begins decaying the learning rate and the loss continues to improve, but much more gradually, ending near 0.3366 by epoch 4.

Qualitatively, the model clearly learned common sentence structure and many high-frequency Spanish content words within only four epochs, but it still repeated tokens,

dropped function words, and occasionally collapsed to short partial phrases. This is the expected behavior as there are not enough runs to produce a strong translator.

## 1.5 (e) Text: Self-attention computation explanation

If the input sentence contains $N_w$ tokens and each token embedding has width $M$, then the learned query and key tensors are $Q, K \in \mathbb{R}^{N_w \times M}$. The matrix product $QK^\top$ therefore has shape $N_w \times N_w$: each entry compares one token's query against another token's key. After row-wise `Softmax`, row $i$ becomes a probability distribution over all source positions, so it tells us how strongly token $i$ attends to every token in the sentence, including itself.

The division by $\sqrt{M}$ prevents the raw dot products from growing too large when the embedding dimension is large. Without that scaling, the logits entering the softmax would have high variance, the softmax would become overly peaked early in training, and the gradients would become unstable or very small. The scaling keeps attention scores in a numerically reasonable range.

## 1.6 (f) Text: Multi-headed attention explanation

With embedding size 256 and 4 attention heads, each head receives a slice of size

$$\texttt{sqkv} = 256/4 = 64.$$

Each head therefore learns its own query, key, and value projections in a 64-dimensional subspace, and the outputs are concatenated afterward.

Multiple heads are better than a single full-width head because different heads can specialize in different relationships at the same time. One head may learn local syntactic alignment, another may focus on subject–verb agreement, and another may lock onto longer-range semantic dependencies. A single head with all 256 channels would have to compress all of those roles into one attention pattern.

## 1.7 (g) Text: TransformerFG vs. TransformerPreLN comparison

In TransformerFG, LayerNorm is applied after each sublayer's residual addition, so the pattern is "self-attention $\rightarrow$ residual add $\rightarrow$ LayerNorm" and then "FFN $\rightarrow$ residual add $\rightarrow$ LayerNorm." In TransformerPreLN, LayerNorm is moved before the self-attention and FFN sublayers, so the residual path itself is more direct.

That placement difference explains the warm-up requirement. TransformerFG is a post-norm design, which is less stable during the earliest optimization steps because gradients must pass through more poorly conditioned transformations before the model has learned useful internal scales. TransformerPreLN keeps the residual pathway cleaner, so gradient flow is more stable from the start and the model can usually train with a fixed learning rate instead of a fragile warm-up schedule.

# 2 Task 2: Vision Transformer Warm-up on CIFAR-10 (15 points)

## 2.1 (a) Code: Script used to run visTransformer

```python
def main() -> None:
    """Train the CIFAR-10 visTransformer and evaluate the best checkpoint."""

    args = parse_args()
    setup_seed(args.seed)
    results_dir = ensure_dir(PROJECT_ROOT / args.results_dir)
    figures_dir = ensure_dir(PROJECT_ROOT / args.figures_dir)
    checkpoint_dir = ensure_dir(results_dir / "checkpoints_visTrans")

    dls = DLStudio(
        dataroot=str(PROJECT_ROOT / args.data_root),
        image_size=(32, 32),
        path_saved_model=str(results_dir / "saved_visTran_model"),
        learning_rate=args.learning_rate,
        epochs=args.epochs,
        batch_size=args.batch_size,
        classes=CIFAR10_CLASSES,
        use_gpu=True,
    )
    device = dls.device

    model = visTransformer(
        dl_studio=dls,
        patch_size=(8, 8),
        embedding_size=args.embedding_size,
        num_basic_encoders=args.num_basic_encoders,
        num_atten_heads=args.num_atten_heads,
        save_checkpoints=False,
    )
    model.to(device)
    patch_embedding_generator = model.PatchEmbeddingGenerator(model).to(device)
    train_loader, test_loader = build_loaders(args.batch_size, PROJECT_ROOT / args.data_root)

    optimizer = torch.optim.Adam(model.parameters(), lr=args.learning_rate, betas=(0.9, 0.99))
    patch_optimizer = torch.optim.Adam(patch_embedding_generator.parameters(), lr=args.learning_rate, betas=(0.9, 0.99))
    criterion = nn.CrossEntropyLoss()

```

```
    training_loss_history: list[float] = []
    checkpoint_scores: list[dict[str, float]] = []
    start_time = time.perf_counter()

    for epoch in range(1, args.epochs + 1):
        model.train()
        patch_embedding_generator.train()
        running_loss = 0.0
        example_count = 0
        for images, labels in train_loader:
            images = images.to(device, non_blocking=True)
            labels = labels.to(device, non_blocking=True)
            optimizer.zero_grad()
            patch_optimizer.zero_grad()
            tokens = make_tokens(model, patch_embedding_generator,
               images)
            logits = model(tokens)
            loss = criterion(logits, labels)
            loss.backward()
            optimizer.step()
            patch_optimizer.step()
            running_loss += loss.item() * labels.shape[0]
            example_count += labels.shape[0]

        mean_train_loss = running_loss / max(example_count, 1)
        training_loss_history.append(mean_train_loss)
        elapsed = time.perf_counter() - start_time
        print(f"epoch {epoch:02d}/{args.epochs} train_loss={
           mean_train_loss:.4f} elapsed={elapsed:.1f}s")

        if epoch % args.checkpoint_freq == 0:
            save_checkpoint(model, patch_embedding_generator,
               checkpoint_dir, epoch)
            checkpoint_eval = evaluate(model,
               patch_embedding_generator, test_loader, device)
            checkpoint_scores.append({"epoch": epoch, "accuracy":
               checkpoint_eval["accuracy"]})
            print(f"checkpoint epoch {epoch}: accuracy={
               checkpoint_eval['accuracy']:.4%}")

    if not checkpoint_scores:
        checkpoint_scores.append({"epoch": args.epochs, "accuracy":
           evaluate(model, patch_embedding_generator, test_loader,
           device)["accuracy"]})
        save_checkpoint(model, patch_embedding_generator,
           checkpoint_dir, args.epochs)

```

```
    best_checkpoint = max(checkpoint_scores, key=lambda item: item["accuracy"])
    load_checkpoint(model, patch_embedding_generator, checkpoint_dir, int(best_checkpoint["epoch"]), device)
    final_eval = evaluate(model, patch_embedding_generator, test_loader, device)

    save_curve(
        training_loss_history,
        title="CIFAR-10 visTransformer Training Loss",
        xlabel="Epoch",
        ylabel="Cross-Entropy Loss",
        output_path=figures_dir / "task2_cifar_loss.png",
        marker="o",
    )
    save_confusion_matrix_figure(
        final_eval["confusion_matrix"],
        CIFAR10_CLASSES,
        figures_dir / "task2_cifar_confusion_matrix.png",
        title="CIFAR-10 visTransformer Confusion Matrix",
    )

    confusion_text_lines = ["          " + " ".join(f"{name:>7s}" for name in CIFAR10_CLASSES)]
    for class_index, class_name in enumerate(CIFAR10_CLASSES):
        row = final_eval["confusion_matrix"][class_index]
        row_text = " ".join(f"{int(value):7d}" for value in row)
        confusion_text_lines.append(f"{class_name:>8s} {row_text}")
    (results_dir / "task2_confusion_matrix.txt").write_text("\n".join(confusion_text_lines), encoding="utf-8")

    summary = {
        "config": vars(args),
        "parameter_count": count_parameters(model) + count_parameters(patch_embedding_generator),
        "training_loss_history": training_loss_history,
        "checkpoint_scores": checkpoint_scores,
```

Listing 2: Task 2 training, checkpointing, and evaluation wrapper around DLStudio's visTransformer

## 2.2 (b) Figure: Training loss curve over 40 epochs

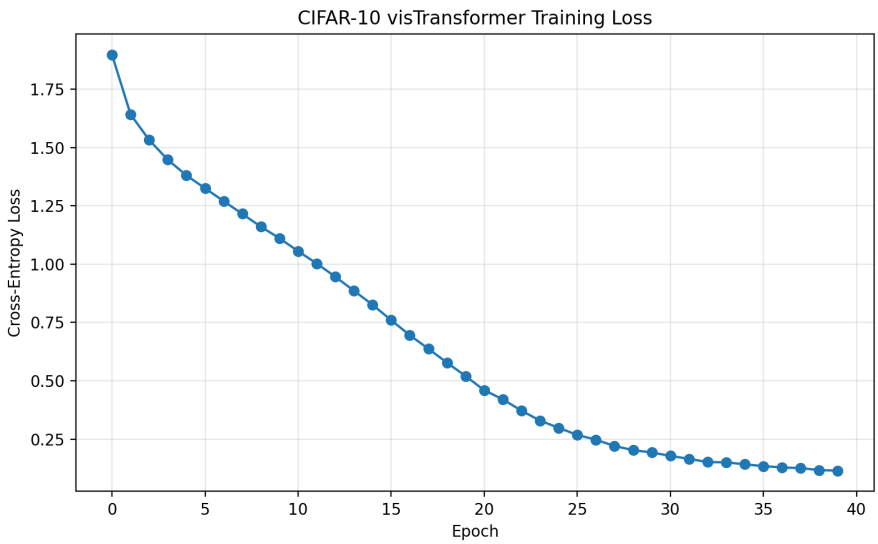



Figure 2: Task 2 CIFAR-10 visTransformer training loss over 40 epochs.

## 2.3 (c) Table: 10×10 confusion matrix on test set

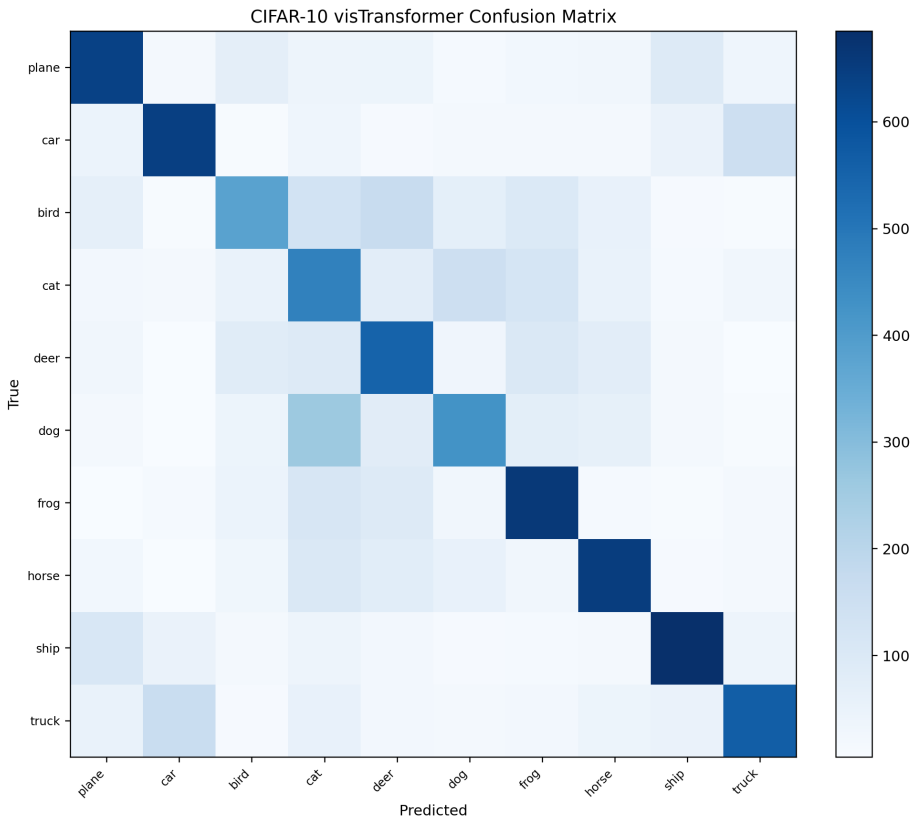



Figure 3: Task 2 confusion matrix for the best saved checkpoint (epoch 20).

Table 2: Task 2 test confusion matrix counts for the best checkpoint (epoch 20).

| True \ Pred. | plane | car | bird | cat | deer | dog | frog | horse | ship | truck |
|---|---|---|---|---|---|---|---|---|---|---|
| plane | 639 | 18 | 71 | 38 | 42 | 13 | 26 | 28 | 90 | 35 |
| car | 47 | 645 | 10 | 36 | 11 | 19 | 16 | 18 | 48 | 150 |
| bird | 66 | 10 | 381 | 131 | 167 | 68 | 99 | 57 | 12 | 9 |
| cat | 23 | 18 | 51 | 472 | 75 | 149 | 120 | 51 | 13 | 28 |
| deer | 27 | 5 | 84 | 94 | 552 | 34 | 103 | 76 | 18 | 7 |
| dog | 18 | 6 | 42 | 262 | 82 | 426 | 74 | 61 | 19 | 10 |
| frog | 5 | 14 | 46 | 112 | 95 | 28 | 659 | 15 | 10 | 16 |
| horse | 25 | 5 | 29 | 101 | 78 | 55 | 28 | 648 | 11 | 20 |
| ship | 110 | 49 | 16 | 39 | 23 | 11 | 14 | 16 | 685 | 37 |
| truck | 51 | 162 | 11 | 58 | 21 | 19 | 23 | 42 | 49 | 564 |

## 2.4 (d) Text: Overall test accuracy

Checkpoint evaluation gave the following accuracies: epoch 10: 55.47%, epoch 20: 56.71%, epoch 30: 55.79%, epoch 40: 56.24%. The best checkpoint was epoch 20, with overall test accuracy **56.71**%.

## 2.5 (e) Text: Patch embedding explanation

For $32 \times 32 \times 3$ CIFAR-10 images and a patch size of $16 \times 16$, each image yields

$$P = (32/16) \times (32/16) = 4$$

patches. Each raw patch contains $16 \times 16 \times 3 = 768$ pixel values before embedding. After the linear projection to embedding width $M$, the patch tensor shape becomes $[4, M]$ if we ignore the batch axis.

## 2.6 (f) Text: Class token explanation

After concatenating one learnable class token with the 4 patch embeddings, the tensor entering the encoder has shape $[5, M]$. The first position corresponds to the class token and the remaining four correspond to image patches.

Because self-attention allows every token to exchange information with every other token at every layer, the class token acts like a learned global summary slot. By the time it emerges from the final encoder, it has repeatedly aggregated evidence from all patches and can therefore serve as a compact representation for the final classification layer.

## 2.7 (g) Text: Positional encoding explanation

Without positional encodings, the transformer would treat the 4 CIFAR-10 patches as an unordered set. Swapping the top-left patch with the bottom-right patch would produce the same attention computation up to permutation, even though the image semantics would change.

Positional encodings inject the notion of spatial location into the token stream. That lets the model learn distinctions such as "sky tends to be above land" or "wheels usually sit below the body of a car." Removing positional encodings would significantly weaken the model's ability to reason about image layout.

# 3 Task 3: Food-101 From Scratch + Attention Visualization (25 points)

## 3.1 (a) Code: Adapted visTransformer for Food-101 subset

```
$env:KMP_DUPLICATE_LIB_OK='TRUE'
conda run -n ece60146 python attention_viz.py ^
    --mode all --epochs 20 --batch-size 64 ^
    --results-dir results_task3 --figures-dir figures
```

Listing 3: Task 3 command used for the final from-scratch Food-20 run

```
class AttentionHead(nn.Module):
    """A single self-attention head that stores its softmax weights.
       """

    def __init__(self, qkv_size: int) -> None:
        super().__init__()
        self.qkv_size = qkv_size
        self.wq = nn.Linear(qkv_size, qkv_size, bias=False)
        self.wk = nn.Linear(qkv_size, qkv_size, bias=False)
        self.wv = nn.Linear(qkv_size, qkv_size, bias=False)
        self.attention_weights: torch.Tensor | None = None

    def forward(self, token_slice: torch.Tensor) -> torch.Tensor:
        """Apply scaled dot-product attention to one embedding slice.
           """

        queries = self.wq(token_slice)
        keys = self.wk(token_slice)
        values = self.wv(token_slice)
        scores = torch.matmul(queries, keys.transpose(1, 2)) / np.
           sqrt(self.qkv_size)
        weights = torch.softmax(scores, dim=-1)
        self.attention_weights = weights.detach()
        return torch.matmul(weights, values)


class SelfAttention(nn.Module):
    """Split embeddings across heads and concatenate the attended
       slices."""

    def __init__(self, embedding_size: int, num_atten_heads: int) ->
       None:
        super().__init__()
        if embedding_size % num_atten_heads != 0:
            raise ValueError("embedding_size must be divisible by
```

```
                num_atten_heads")
31          self.embedding_size = embedding_size
32          self.num_atten_heads = num_atten_heads
33          self.qkv_size = embedding_size // num_atten_heads
34          self.attention_heads = nn.ModuleList([AttentionHead(self.
               qkv_size) for _ in range(num_atten_heads)])
35
36      def forward(self, tokens: torch.Tensor, collect_attention: bool =
            False):
37          """Return attended tokens and optionally the stacked
               attention maps."""
38
39          head_outputs = []
40          for head_index, head in enumerate(self.attention_heads):
41              token_slice = tokens[:, :, head_index * self.qkv_size : (
                   head_index + 1) * self.qkv_size]
42              head_outputs.append(head(token_slice))
43          concatenated = torch.cat(head_outputs, dim=-1)
44          if not collect_attention:
45              return concatenated
46          attention_maps = torch.stack([head.attention_weights for head
                in self.attention_heads], dim=1)
47          return concatenated, attention_maps
48
49
50  class BasicEncoder(nn.Module):
51      """A residual self-attention block followed by an FFN."""
52
53      def __init__(self, embedding_size: int, num_atten_heads: int) ->
           None:
54          super().__init__()
55          self.self_attention = SelfAttention(embedding_size,
               num_atten_heads)
56          self.norm1 = nn.LayerNorm(embedding_size)
57          self.ffn = nn.Sequential(
58              nn.Linear(embedding_size, 4 * embedding_size),
59              nn.ReLU(inplace=True),
60              nn.Linear(4 * embedding_size, embedding_size),
61          )
62          self.norm2 = nn.LayerNorm(embedding_size)
63
64      def forward(self, tokens: torch.Tensor, collect_attention: bool =
            False):
65          """Forward through one encoder block."""
66
67          if collect_attention:
68              attended, attention_maps = self.self_attention(tokens,
```

```
                collect_attention=True)
        else:
            attended = self.self_attention(tokens, collect_attention=
                False)
            attention_maps = None
        tokens = self.norm1(tokens + attended)
        tokens = self.norm2(tokens + self.ffn(tokens))
        if collect_attention:
            return tokens, attention_maps
        return tokens


class MasterEncoder(nn.Module):
    """Stack multiple BasicEncoder layers and optionally return
       attention maps."""

    def __init__(self, num_layers: int, embedding_size: int,
       num_atten_heads: int) -> None:
        super().__init__()
        self.layers = nn.ModuleList(
            [BasicEncoder(embedding_size=embedding_size,
                num_atten_heads=num_atten_heads) for _ in range(
                num_layers)]
        )

    def forward(self, tokens: torch.Tensor, collect_attention: bool =
        False):
        """Run the full encoder stack."""

        attention_weights: list[torch.Tensor] = []
        output = tokens
        for layer in self.layers:
            if collect_attention:
                output, layer_attention = layer(output,
                    collect_attention=True)
                attention_weights.append(layer_attention)
            else:
                output = layer(output, collect_attention=False)
        if collect_attention:
            return output, attention_weights
        return output


class FoodVisTransformer(nn.Module):
    """Small ViT-style classifier for the 20-class Food-101 subset.
       """

```

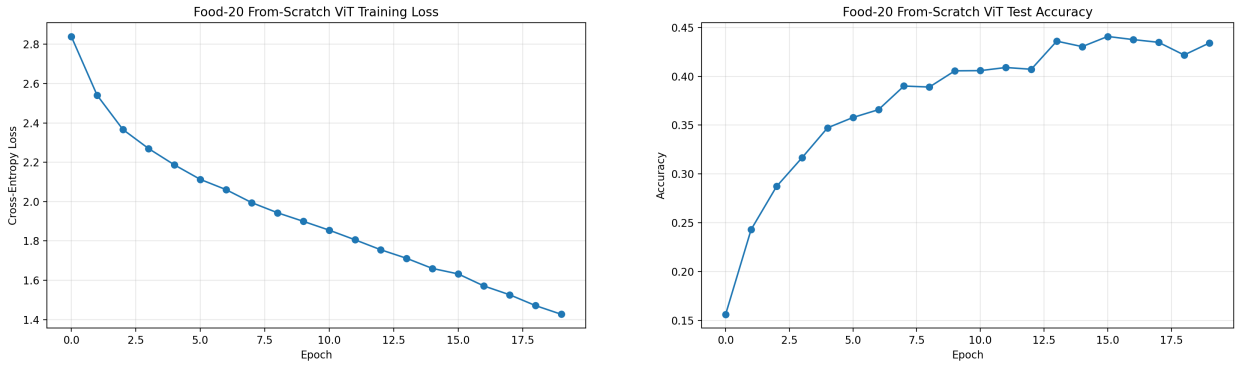
```
    def __init__(
        self,
        image_size: int = 64,
        patch_size: int = 8,
        embedding_size: int = 128,
        num_basic_encoders: int = 4,
        num_atten_heads: int = 4,
        num_classes: int = 20,
    ) -> None:
        super().__init__()
        self.image_size = image_size
        self.patch_size = patch_size
        self.embedding_size = embedding_size
        self.num_basic_encoders = num_basic_encoders
        self.num_atten_heads = num_atten_heads
        self.num_classes = num_classes
        self.patch_embedder = PatchEmbeddingGenerator(image_size,
           patch_size, embedding_size)
        self.class_token = nn.Parameter(torch.randn(1, 1,
           embedding_size) * 0.02)
        self.master_encoder = MasterEncoder(num_basic_encoders,
           embedding_size, num_atten_heads)
        self.classifier = nn.Sequential(
            nn.LayerNorm(embedding_size),
            nn.Linear(embedding_size, 256),
            nn.GELU(),
            nn.Dropout(0.1),
            nn.Linear(256, num_classes),
        )

    def prepare_tokens(self, images: torch.Tensor) -> torch.Tensor:
        """Build `[cls] + patch_embeddings` tokens from raw images.
           """

        patch_embeddings = self.patch_embedder(images)
        class_tokens = self.class_token.expand(images.shape[0], -1,
           -1)
        return torch.cat((class_tokens, patch_embeddings), dim=1)

    def forward_tokens(self, tokens: torch.Tensor, collect_attention:
        bool = False):
        """Run the encoder stack on prebuilt tokens."""

        if collect_attention:
            encoded, attention_weights = self.master_encoder(tokens,
               collect_attention=True)
            logits = self.classifier(encoded[:, 0])
```

```
            return logits, attention_weights
        encoded = self.master_encoder(tokens, collect_attention=False)
        return self.classifier(encoded[:, 0])

    def forward(self, x: torch.Tensor, collect_attention: bool = False):
        """Accept either images `[B, 3, H, W]` or token sequences `[B, S, M]`."""

        if x.dim() == 4:
            tokens = self.prepare_tokens(x)
        elif x.dim() == 3:
            tokens = x
        else:
            raise ValueError("Expected images with 4 dims or token sequences with 3 dims.")
        return self.forward_tokens(tokens, collect_attention=collect_attention)


def extract_attention_weights(model: FoodVisTransformer, input_tensor: torch.Tensor) -> list[torch.Tensor]:
    """Return one `[heads, seq_len, seq_len]` tensor per encoder layer.

    The autograder supplies a token tensor shaped like `[1, seq_len, embedding_size]`,
    so this function uses the encoder path directly and returns the first batch item.
    """

    device = next(model.parameters()).device
    model.eval()
    with torch.no_grad():
        _, attention_weights = model(input_tensor.to(device), collect_attention=True)
    return [layer_weights[0].detach().cpu() for layer_weights in attention_weights]
```

Listing 4: Modified attention stack, encoder, model, and required autograder hook in `attention_viz.py`

The adapted model uses the exact 20-class Food-101 subset from the assignment, resizes images to $64 \times 64$, and uses $8 \times 8$ patches. Therefore each image is converted to 64 patch tokens, and after prepending the class token the sequence length is 65, which produces a $65 \times 65$ attention map per head. The final classifier outputs 20 logits. The final from-scratch model had 649,492 learnable parameters.

## 3.2 (b) Figure: Training loss curve




(a) Training loss

(b) Test accuracy per epoch

Figure 4: Task 3 from-scratch Food-20 training curves.

## 3.3 (c) Text: Test accuracy and training observations

The model converged steadily from 15.62% test accuracy after epoch 1 to a best of **44.08**% at epoch 16. Since random guessing over 20 classes would only give 5%, the trained model is clearly learning meaningful category structure even though it is far below the pretrained ViT in Task 4.

## 3.4 (d) Code: Modified AttentionHead, SelfAttention, BasicEncoder, and MasterEncoder

```python
class AttentionHead(nn.Module):
    """A single self-attention head that stores its softmax weights.
       """

    def __init__(self, qkv_size: int) -> None:
        super().__init__()
        self.qkv_size = qkv_size
        self.wq = nn.Linear(qkv_size, qkv_size, bias=False)
        self.wk = nn.Linear(qkv_size, qkv_size, bias=False)
        self.wv = nn.Linear(qkv_size, qkv_size, bias=False)
        self.attention_weights: torch.Tensor | None = None

    def forward(self, token_slice: torch.Tensor) -> torch.Tensor:
        """Apply scaled dot-product attention to one embedding slice.
           """

        queries = self.wq(token_slice)
        keys = self.wk(token_slice)
        values = self.wv(token_slice)
```

```
        scores = torch.matmul(queries, keys.transpose(1, 2)) / np.sqrt(self.qkv_size)
        weights = torch.softmax(scores, dim=-1)
        self.attention_weights = weights.detach()
        return torch.matmul(weights, values)


class SelfAttention(nn.Module):
    """Split embeddings across heads and concatenate the attended slices."""

    def __init__(self, embedding_size: int, num_atten_heads: int) -> None:
        super().__init__()
        if embedding_size % num_atten_heads != 0:
            raise ValueError("embedding_size must be divisible by num_atten_heads")
        self.embedding_size = embedding_size
        self.num_atten_heads = num_atten_heads
        self.qkv_size = embedding_size // num_atten_heads
        self.attention_heads = nn.ModuleList([AttentionHead(self.qkv_size) for _ in range(num_atten_heads)])

    def forward(self, tokens: torch.Tensor, collect_attention: bool = False):
        """Return attended tokens and optionally the stacked attention maps."""

        head_outputs = []
        for head_index, head in enumerate(self.attention_heads):
            token_slice = tokens[:, :, head_index * self.qkv_size : (head_index + 1) * self.qkv_size]
            head_outputs.append(head(token_slice))
        concatenated = torch.cat(head_outputs, dim=-1)
        if not collect_attention:
            return concatenated
        attention_maps = torch.stack([head.attention_weights for head in self.attention_heads], dim=1)
        return concatenated, attention_maps


class BasicEncoder(nn.Module):
    """A residual self-attention block followed by an FFN."""

    def __init__(self, embedding_size: int, num_atten_heads: int) -> None:
        super().__init__()
```

```
55          self.self_attention = SelfAttention(embedding_size,
                num_atten_heads)
56          self.norm1 = nn.LayerNorm(embedding_size)
57          self.ffn = nn.Sequential(
58              nn.Linear(embedding_size, 4 * embedding_size),
59              nn.ReLU(inplace=True),
60              nn.Linear(4 * embedding_size, embedding_size),
61          )
62          self.norm2 = nn.LayerNorm(embedding_size)
63
64      def forward(self, tokens: torch.Tensor, collect_attention: bool =
            False):
65          """Forward through one encoder block."""
66
67          if collect_attention:
68              attended, attention_maps = self.self_attention(tokens,
                    collect_attention=True)
69          else:
70              attended = self.self_attention(tokens, collect_attention=
                    False)
71              attention_maps = None
72          tokens = self.norm1(tokens + attended)
73          tokens = self.norm2(tokens + self.ffn(tokens))
74          if collect_attention:
75              return tokens, attention_maps
76          return tokens
77
78
79  class MasterEncoder(nn.Module):
80      """Stack multiple BasicEncoder layers and optionally return
            attention maps."""
81
82      def __init__(self, num_layers: int, embedding_size: int,
            num_atten_heads: int) -> None:
83          super().__init__()
84          self.layers = nn.ModuleList(
85              [BasicEncoder(embedding_size=embedding_size,
                    num_atten_heads=num_atten_heads) for _ in range(
                    num_layers)]
86          )
87
88      def forward(self, tokens: torch.Tensor, collect_attention: bool =
            False):
89          """Run the full encoder stack."""
90
91          attention_weights: list[torch.Tensor] = []
92          output = tokens
```

```
        for layer in self.layers:
            if collect_attention:
                output, layer_attention = layer(output,
                    collect_attention=True)
                attention_weights.append(layer_attention)
            else:
                output = layer(output, collect_attention=False)
        if collect_attention:
            return output, attention_weights
        return output


class FoodVisTransformer(nn.Module):
    """Small ViT-style classifier for the 20-class Food-101 subset.
       """
```

Listing 5: AttentionHead with stored weights, SelfAttention, BasicEncoder, and MasterEncoder with optional attention collection

## 3.5 (e) Text: Description of attention extraction modifications

I added an `attention_weights` instance attribute to `AttentionHead` immediately after the row-wise softmax. Then `SelfAttention` stacks those stored maps into a tensor of shape [batch, heads, seq, seq], and `BasicEncoder` / `MasterEncoder` thread those tensors upward only when `collect_attention=True`. The top-level helper `extract_attention_weights(model, input_tensor)` runs a no-grad forward pass and returns a list of [heads, seq, seq] tensors, one per encoder layer, exactly as required by the autograder.

These changes do not interfere with training because they only detach and read the softmax weights. The classifier path remains unchanged: the class token still summarizes the encoded sequence and the final linear head still predicts one of the 20 food categories.

### 3.6 (f) Figure: Part A — 4 images with full heatmap and spatial overlay

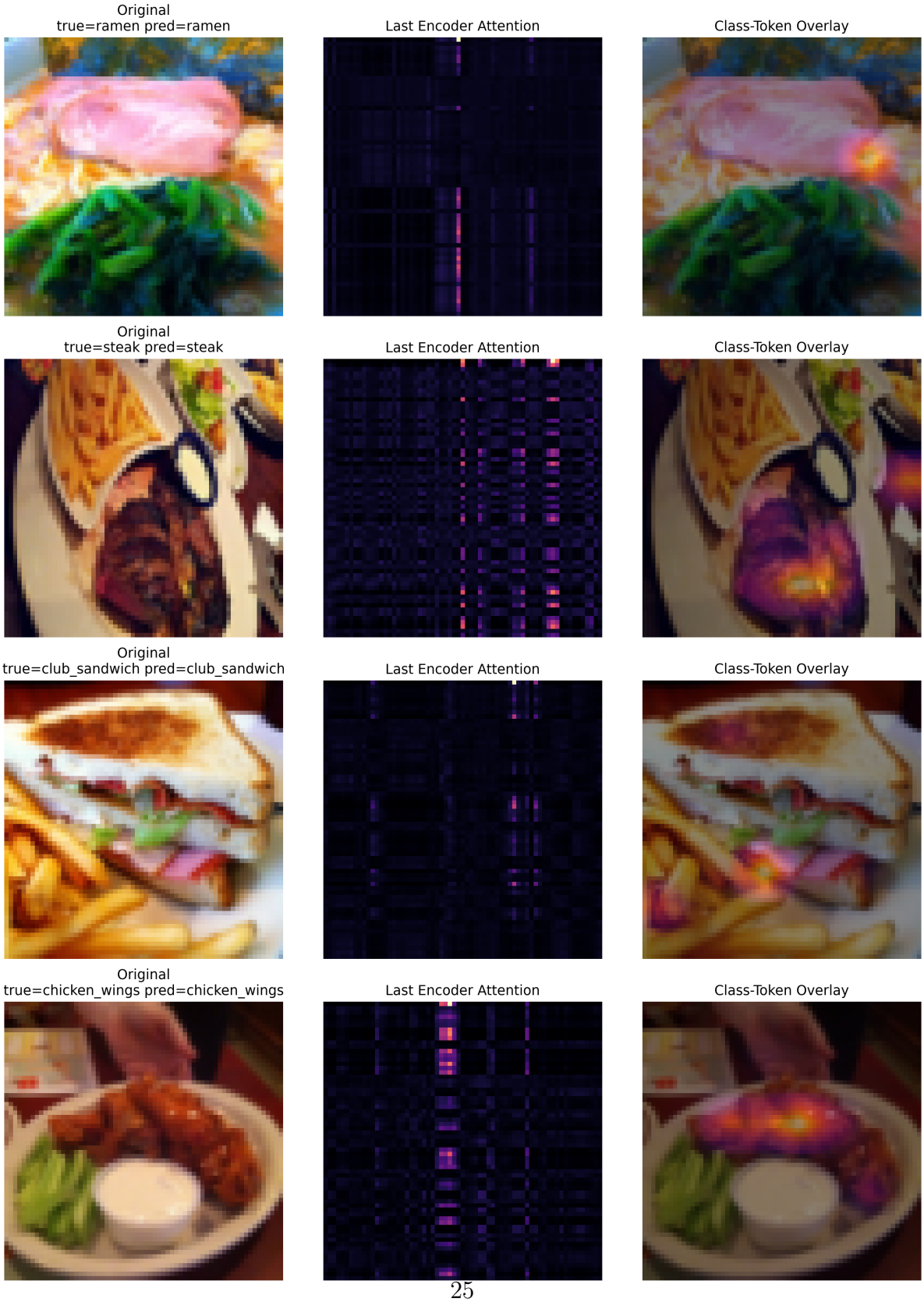





Figure 5: Task 3 Part A. For each correctly classified image: original image, full $65 \times 65$ last-layer attention map for one head, and class-token spatial overlay.

## 3.7 (g) Text: Part A — Per-image attention analysis

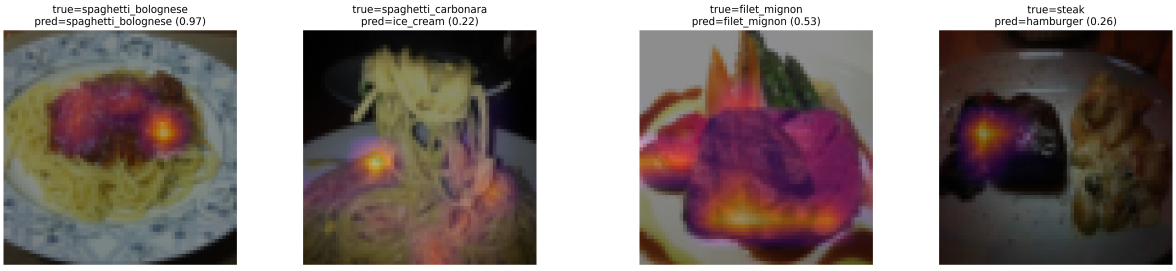
**Ramen.** The overlay concentrates on the bright garnish and pork region on the right side of the bowl while still covering part of the noodles and broth. That suggests the class token is using a visually distinctive topping pattern rather than uniformly attending to the entire bowl.

**Steak.** The class token focuses on the browned meat surface in the lower part of the plate and mostly ignores the fries. That is a sensible decision boundary because the seared meat texture is more class-specific than the side dish.

**Club sandwich.** The strongest attention falls near the sandwich interior and plate garnish region, where the layered bread, filling, and color contrast are most obvious. The fries occupy a large area in the image, but the overlay treats them as secondary context.

**Chicken wings.** The map emphasizes the glossy wing surface across the upper-middle and right side of the plate while largely ignoring the dipping sauce cup and lettuce garnish. The model appears to be using the sauce-coated wing texture and curved wing shapes as the main evidence.

## 3.8 (h) Figure: Part B — 2 pairs with side-by-side overlays




(a) Spaghetti bolognese vs. spaghetti carbonara (b) Filet mignon vs. steak

Figure 6: Task 3 Part B side-by-side class-token attention overlays for two visually similar category pairs.

## 3.9 (i) Text: Part B — Comparative analysis per pair

**Spaghetti bolognese vs. spaghetti carbonara.** For the bolognese image, the class token concentrates on the red meat-sauce mound at the center-right, which is exactly the feature that differentiates it from plain or cream-based pasta dishes. For the carbonara image, the attention centers on the pale creamy region and glossy highlights rather than on clear pasta structure, and the model misclassifies it as `ice_cream`. That failure suggests the low-color-contrast, smooth cream texture may be confusing the model more than the actual noodle layout. The two overlays confirm the model is responding to sauce color and texture rather than to the shape of the noodles themselves.

**Filet mignon vs. steak.** The filet mignon example receives broad attention across the meat surface, especially the lower seared edge and central body of the cut, and it is classified correctly. In contrast, the steak example is misclassified as `hamburger`, and the overlay locks onto a dark rectangular slab of meat while paying relatively little attention to the surrounding potatoes. That makes sense: the attended region looks somewhat patty-like, so the model appears to be collapsing the two red-meat categories when the cut shape is ambiguous. A clearer plating or a different camera angle might have produced the correct prediction.

# 4 Task 4: Pretrained ViT Fine-tuning on Food-101 Subset (45 points)

## 4.1 (a) Code: Fine-tuning script for ViT-Base on Food-101 subset

```python
def build_loaders(batch_size: int, data_root: str | Path) -> tuple[DataLoader, DataLoader]:
    """Create train and test loaders with ImageNet-style transforms."""

    train_transform = T.Compose(
        [
            T.RandomResizedCrop(224),
            T.RandomHorizontalFlip(),
            T.ToTensor(),
            T.Normalize(IMAGENET_MEAN, IMAGENET_STD),
        ]
    )
    test_transform = T.Compose(
        [
            T.Resize(256),
            T.CenterCrop(224),
            T.ToTensor(),
            T.Normalize(IMAGENET_MEAN, IMAGENET_STD),
        ]
    )
    train_bundle = make_food20("train", train_transform, root=data_root)
    test_bundle = make_food20("test", test_transform, root=data_root)
    train_loader = DataLoader(train_bundle.dataset, batch_size=batch_size, shuffle=True, num_workers=4, pin_memory=True)
    test_loader = DataLoader(test_bundle.dataset, batch_size=batch_size, shuffle=False, num_workers=4, pin_memory=True)
    return train_loader, test_loader


def maybe_freeze_backbone(model: nn.Module, freeze: bool) -> None:
    """Optionally freeze the patch embed and transformer blocks."""

    for parameter in model.patch_embed.parameters():
        parameter.requires_grad = not freeze
    for block in model.blocks:
        for parameter in block.parameters():
            parameter.requires_grad = not freeze


```

```python
def evaluate(model: nn.Module, loader: DataLoader, device: torch.device) -> dict[str, Any]:
    """Evaluate a classifier and keep 16 examples for the prediction grid."""

    model.eval()
    correct = 0
    total = 0
    dataset_size = len(loader.dataset)
    target_positions = set(np.linspace(0, dataset_size - 1, 16, dtype=int).tolist())
    global_index = 0
    sampled_images: list[torch.Tensor] = []
    sampled_predictions: list[str] = []
    sampled_truths: list[str] = []
    with torch.no_grad():
        for images, labels in loader:
            images = images.to(device, non_blocking=True)
            labels = labels.to(device, non_blocking=True)
            logits = model(images)
            predictions = torch.argmax(logits, dim=1)
            correct += (predictions == labels).sum().item()
            total += labels.numel()
            for image, prediction, truth in zip(images.cpu(), predictions.cpu(), labels.cpu()):
                if global_index in target_positions:
                    sampled_images.append(image)
                    sampled_predictions.append(FOOD20_CLASSES[int(prediction)])
                    sampled_truths.append(FOOD20_CLASSES[int(truth)])
                global_index += 1
    return {
        "accuracy": correct / max(total, 1),
        "sampled_images": sampled_images,
        "sampled_predictions": sampled_predictions,
        "sampled_truths": sampled_truths,
    }
```

Listing 6: Task 4 data transforms, evaluation helper, and 16-image grid sampling logic

```python
def main() -> None:
    """Fine-tune the pretrained ViT-Base model and save all report artifacts."""

    args = parse_args()
    setup_seed(args.seed)
    results_dir = ensure_dir(PROJECT_ROOT / args.results_dir)
    figures_dir = ensure_dir(PROJECT_ROOT / args.figures_dir)
```

```
    checkpoint_path = results_dir / "finetuned_vit_best.pt"

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    train_loader, test_loader = build_loaders(args.batch_size, PROJECT_ROOT / args.data_root)
    model = timm.create_model("vit_base_patch16_224", pretrained=True, num_classes=len(FOOD20_CLASSES)).to(device)

    criterion = nn.CrossEntropyLoss()
    train_loss_history: list[float] = []
    validation_accuracy_history: list[float] = []
    best_accuracy = -1.0
    start_time = time.perf_counter()

    for epoch in range(1, args.epochs + 1):
        maybe_freeze_backbone(model, freeze=epoch <= args.freeze_epochs)
        optimizer = torch.optim.AdamW(
            [parameter for parameter in model.parameters() if parameter.requires_grad],
            lr=args.learning_rate,
            weight_decay=args.weight_decay,
        )

        model.train()
        running_loss = 0.0
        example_count = 0
        for images, labels in train_loader:
            images = images.to(device, non_blocking=True)
            labels = labels.to(device, non_blocking=True)
            optimizer.zero_grad()
            logits = model(images)
            loss = criterion(logits, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item() * labels.shape[0]
            example_count += labels.shape[0]

        mean_loss = running_loss / max(example_count, 1)
        validation = evaluate(model, test_loader, device)
        validation_accuracy = validation["accuracy"]
        train_loss_history.append(mean_loss)
        validation_accuracy_history.append(validation_accuracy)
        elapsed = time.perf_counter() - start_time
        print(
            f"epoch {epoch:02d}/{args.epochs} train_loss={mean_loss
```

```
                :.4f} "
            f"val_accuracy={validation_accuracy:.4%} elapsed={elapsed
                :.1f}s"
        )

        if validation_accuracy > best_accuracy:
            best_accuracy = validation_accuracy
            torch.save(
                {
                    "model_state": model.state_dict(),
                    "config": vars(args),
                    "best_accuracy": best_accuracy,
                },
                checkpoint_path,
            )

    checkpoint = torch.load(checkpoint_path, map_location=device)
    model.load_state_dict(checkpoint["model_state"])
    model.to(device)
    final_eval = evaluate(model, test_loader, device)

    save_dual_curve(
        train_loss_history,
        validation_accuracy_history,
        first_label="Training Loss",
        second_label="Validation Accuracy",
        title="ViT-Base Fine-Tuning Curves",
        xlabel="Epoch",
        ylabel="Value",
        output_path=figures_dir / "task4_vit_curves.png",
    )
    save_prediction_grid(
        final_eval["sampled_images"],
        final_eval["sampled_predictions"],
        final_eval["sampled_truths"],
        figures_dir / "task4_vit_prediction_grid.png",
        mean=IMAGENET_MEAN,
        std=IMAGENET_STD,
    )

    summary = {
        "config": vars(args),
        "train_loss_history": train_loss_history,
        "validation_accuracy_history": validation_accuracy_history,
        "best_accuracy": best_accuracy,
        "test_accuracy": final_eval["accuracy"],
        "parameter_count": count_parameters(model),
```

Listing 7: Task 4 ViT-Base fine-tuning loop and checkpoint selection

## 4.2 (b) Figure: Training loss and validation accuracy curves

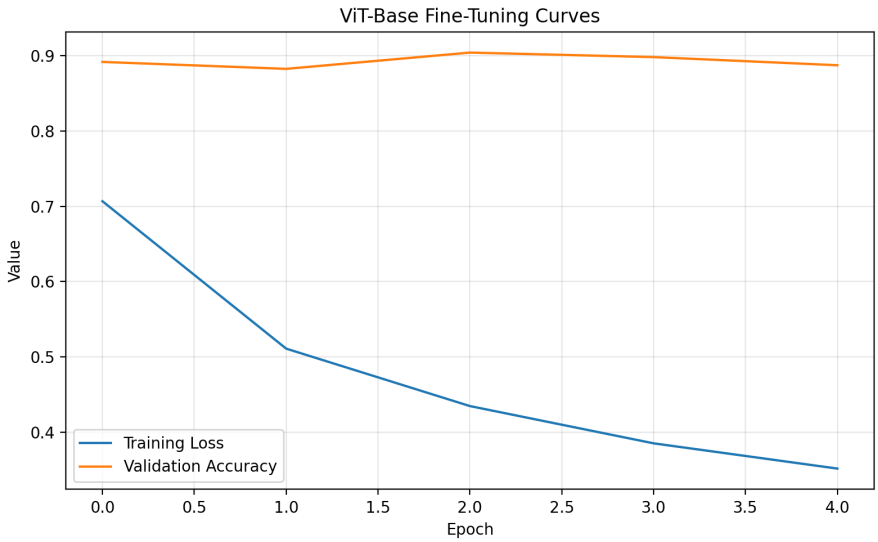



Figure 7: Task 4 training loss and validation accuracy curves over 5 fine-tuning epochs.

## 4.3 (c) Text: Test accuracy and parameter count

The best fine-tuned ViT-Base checkpoint reached **90.40**% accuracy on the Food-20 test split, with the peak occurring at epoch 3. The model has **85,814,036** learnable parameters.

## 4.4 (d) Figure: 4×4 image grid with predictions

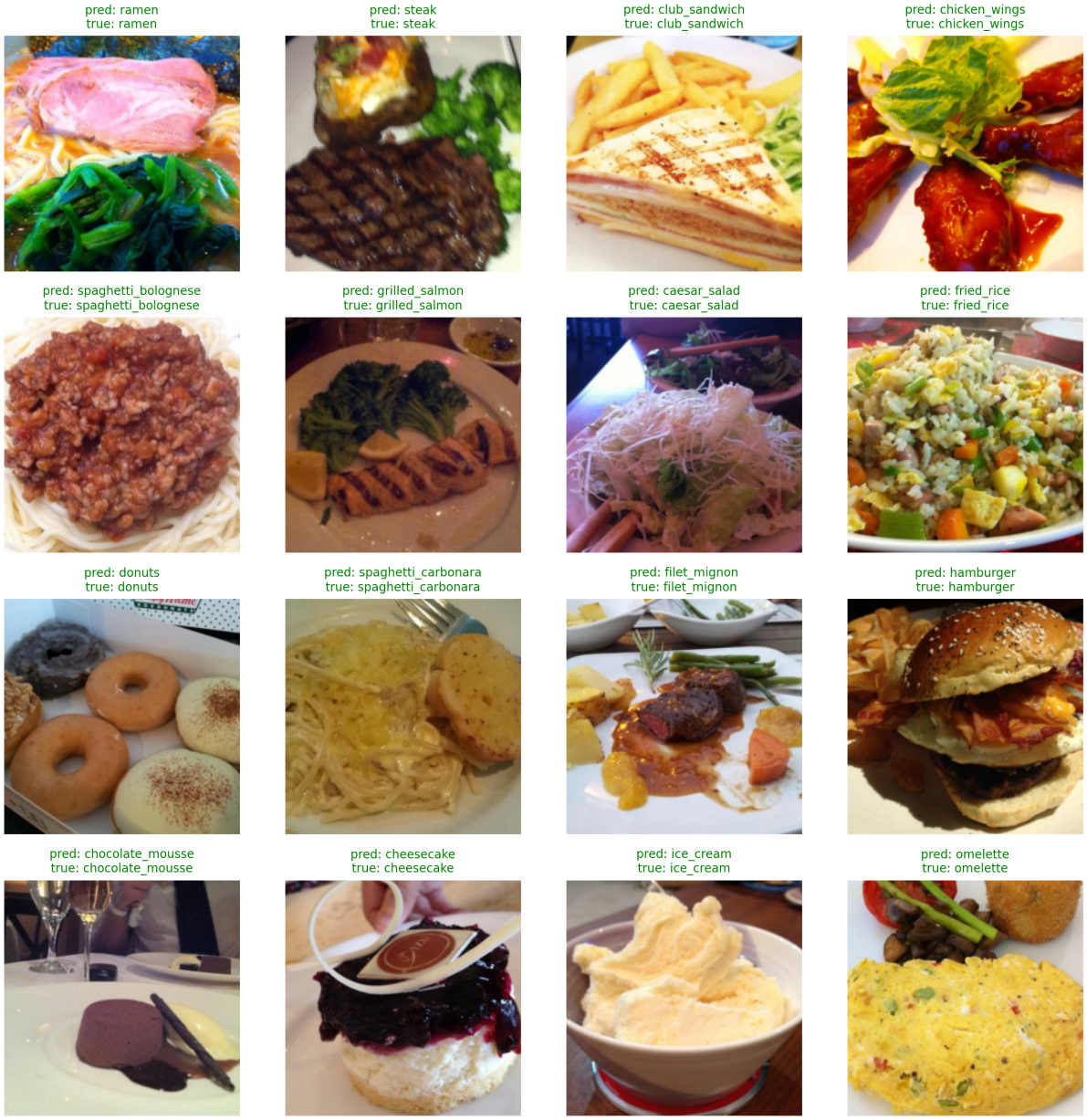



Figure 8: Task 4 4×4 test-image grid with predicted and true labels. Incorrect predictions are shown in red; the saved sample here is entirely correct.

## 4.5 (e) Table: Two-model comparison table

Table 3: Comparison of the from-scratch Food-20 model and the pretrained ViT-Base.

| Model | Input Size | Pretrained? | Accuracy | Params |
|---|---|---|---|---|
| `visTransformer` from Task 3 | $64^2$ | No | 44.08% | 649,492 |
| ViT-Base fine-tuned from Task 4 | $224^2$ | Yes | 90.40% | 85,814,036 |

## 4.6 (f) Text: Accuracy gap discussion

The accuracy gap is $90.40\% - 44.08\% = 46.32$ percentage points. All three listed factors matter: the pretrained model is larger, it sees higher-resolution inputs, and it starts from pretrained weights. However, I believe pretraining contributes the most. The from-scratch Food-20 transformer already reaches 44% with only about 0.65M parameters and $64 \times 64$ images, which means the task is learnable from the subset. The dramatic jump to 90% is hard to explain by model size or input resolution alone; the pretrained weights give the model a mature feature hierarchy and much better token-to-token relational priors before fine-tuning even begins.

## 4.7 (g) Text: Model capacity and data requirements discussion

ViT-Base has roughly 85.8M trainable parameters in this run, compared with about 0.65M for the Task 3 model. If I trained ViT-Base from scratch on only 15,000 Food-20 training images, I would expect severe overfitting and optimization difficulty. Unlike a CNN, a ViT has weak built-in inductive bias for locality, so a large randomly initialized transformer usually needs massive pretraining data to learn stable patch representations and inter-patch relationships. In other words, a huge uninitialized ViT would likely do worse than the much smaller Task 3 model because the data budget is too small for its capacity.

## 4.8 (h) Text: Connection to language model pretraining

This homework made the pretraining idea very concrete. In vision, ImageNet pretraining teaches the transformer reusable structure such as edges, textures, part-whole relationships, and category-level semantics. Fine-tuning on Food-20 then only needs to specialize those already-useful features to the target classes.

The language-model analogy is direct. BERT and GPT are pretrained on very large generic text corpora so they learn syntax, word meaning, long-range context, and token prediction structure before a downstream task is introduced. Fine-tuning on a smaller labeled dataset works well because the model is not learning language from scratch anymore; it is just adapting a rich prior to a narrower objective. ViT on ImageNet and language models on large corpora are the same story in two different modalities.