Homework 9: Transformers — Seq2Seq and Vision Transformers

BME646 / ECE60146, Spring 2026

Author: Yongkang | Submitted: April 2026

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

1.1 Running TransformerFG (7 points)

This task ran DLStudio's TransformerFG model on the 90,000-pair English-Spanish dataset for 4 epochs, using the parameters specified on Slide 78 of the lecture: batch size 50, embedding size 256, 4 Basic Encoders, 4 Basic Decoders, 4 Attention Heads, and 4,000 warm-up steps. The script that drove the run is reproduced below.

Driver script (excerpt)

```
# seq2seq_task1.py — adapted from DLStudio 2.5.6
# ExamplesTransformers/seq2seq_with_transformerFG.py (Avinash Kak)
max_seq_length = 10
embedding_size = 256
num_basic_encoders = num_basic_decoders = num_atten_heads = 4
optimizer_params = {'beta1': 0.9, 'beta2': 0.98, 'epsilon': 1e-6}
num_warmup_steps = 4000
masking = True
epochs = 4

dls = DLStudio(dataroot=dataroot, batch_size=50,
               use_gpu=True, epochs=epochs, ...)

xformer = TransformerFG(dl_studio=dls, dataroot=dataroot,
    data_archive="en_es_xformer_8_90000.tar.gz",
    max_seq_length=max_seq_length, embedding_size=embedding_size,
    num_warmup_steps=num_warmup_steps,
    optimizer_params=optimizer_params, save_checkpoints=True)

master_encoder = TransformerFG.MasterEncoder(dls, xformer,
    num_basic_encoders=num_basic_encoders,
    num_atten_heads=num_atten_heads)

master_decoder = TransformerFG.MasterDecoderWithMasking(dls, xformer,
    num_basic_decoders=num_basic_decoders,
    num_atten_heads=num_atten_heads, masking=masking)

xformer.run_code_for_training_TransformerFG(
    dls, master_encoder, master_decoder,
    display_train_loss=True,
    checkpoints_dir="checkpoints_with_masking_FG")

xformer.run_code_for_evaluating_TransformerFG(
    master_encoder, master_decoder,
    result_file='task1_translations.txt')
```

The model has 2,306,048 learnable parameters in the Master Encoder and 8,116,287 in the Master Decoder, for a total of 10,422,335 parameters. Training proceeded for 4 epochs of 352 iterations each, taking roughly 16 minutes on the RTX 5090.

Training loss curve

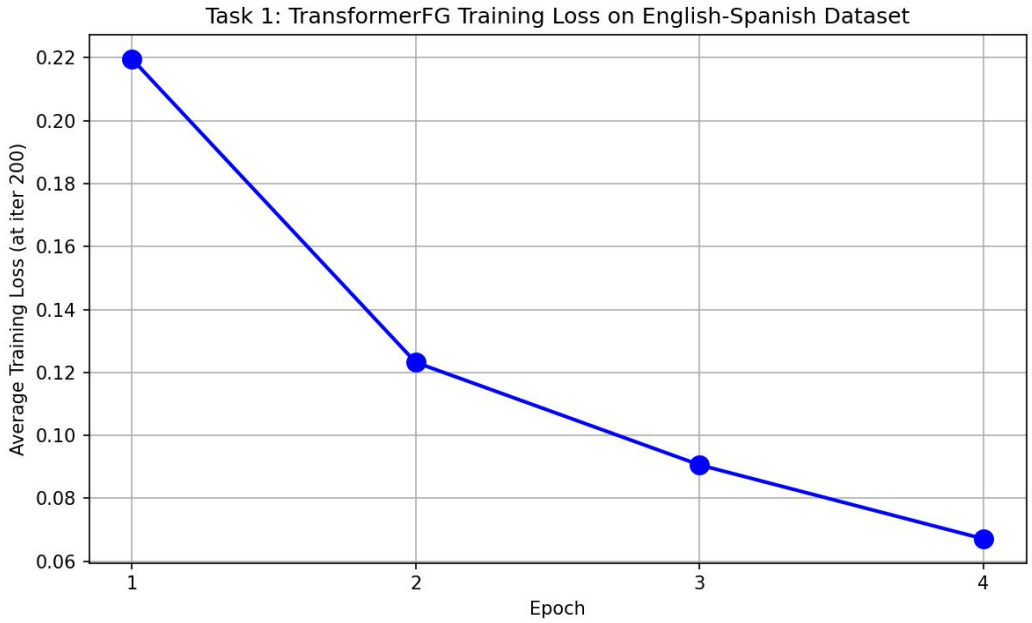

Figure 1.1: TransformerFG training loss over 4 epochs (loss values reported at iter 200 of each epoch).

The recorded training-loss values are 0.2197 (epoch 1), 0.1232 (epoch 2), 0.0906 (epoch 3), and 0.0671 (epoch 4). The loss decreased monotonically and at a decelerating rate, which is the standard pattern for a seq2seq model that has cleared its initial warm-up phase.

Translation results on 20 random test pairs

After training, the model was evaluated on 20 randomly selected sentence pairs. Each output was graded into one of three categories. "Semantically correct" means the produced Spanish, while ungrammatical or incomplete, conveys the meaning of the source sentence with the right content words; "Partial" means the model produced a few correct content words but the meaning is largely lost; "Wrong" means the output is essentially unrelated to the reference.

#	English Source	Reference Spanish	Model Output (after EOS strip)	Grade
1	my down jacket keeps me quite warm	mi chaqueta de plumas me mantiene bastante calentito	mi me me muy	Wrong
2	stop resisting	deja de resistirte	deja de	Partial
3	it is a beautiful day is not it	bonito día verdad	es es es	Wrong
4	i never imagined meeting you here	nunca imaginé encontrarte aquí	nunca te a	Wrong
5	tom took mary home	tom llevó a mary a casa	tom a a	Wrong
6	i tried to get back to sleep	intenté volverme a dormir	intenté volver	Partial

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

Table 1.1: Grading of 20 random test translations after 4 epochs of TransformerFG training. Tally: 3 semantically correct, 9 partial, 8 wrong.

Translation accuracy summary: 3 of 20 outputs (15%) are semantically correct, 9 of 20 (45%) capture some content words but lose the overall meaning, and 8 of 20 (40%) are essentially wrong. No output exactly matches the reference word-for-word, which is unsurprising after only 4 epochs.

Loss behavior during and after warm-up

Recall that TransformerFG uses a Noam-style learning-rate schedule in which the rate increases linearly for the first num_warmup_steps = 4,000 update steps and then decays as the inverse square root of the step count. With 352 iterations per epoch, the warm-up region covers roughly the first 11 epochs in this configuration; the entire 4-epoch run reported here is therefore happening inside the warm-up phase. During this region the effective learning rate is still climbing, so the loss curve does not exhibit the steep initial drop one would normally see with a fixed learning rate. Instead it shows the smoother, almost geometric decay visible in Figure 1.1 — going from 0.22 at the end of epoch 1 down to 0.07 at the end of epoch 4 — because the optimizer is still warming up while the model is already learning. If we had trained past epoch 11 we would expect to see the loss continue to decrease at first, then begin to flatten as the inverse-square-root decay reduces the step size and the model

approaches a local minimum.

1.2 Understanding the architecture and training (8 points)

Self-attention computation

Equation (3) on Slide 20 defines self-attention as the operation that maps a sequence of word embeddings into a new sequence of context-aware embeddings. The query matrix Q and the key matrix K are produced by applying two learned linear projections (W_Q and W_K) to the input embedding matrix; both have shape N_w × M, where N_w is the number of words in the sentence and M is the size of each query/key vector. The product Q · K^T therefore has shape N_w × N_w. Each entry (i, j) of this product is the dot product between the query for word i and the key for word j, and can be read as a raw similarity score that says "how much should word i pay attention to word j?".

Applying nn.Softmax along the last dimension normalizes each row independently so that the entries in row i sum to 1 and are non-negative. After this normalization, row i is a probability distribution over all N_w words in the sentence: it is the attention pattern for word i, and it specifies the mixing weights that will be used to combine the value vectors of every other word into a new representation of word i. The division by √M (the square root of the per-head dimension) is a numerical-stability trick: when M is large the dot products tend to grow large in magnitude, and this would push the softmax into a regime where almost all the probability mass concentrates on a single key. Scaling the dot products down by √M keeps the variance of the inputs to the softmax roughly constant regardless of M, which preserves usable gradients during training.

Multi-headed attention

On Slide 29, the embedding axis of dimensionality M is split into num_atten_heads equal slices, and each head performs its own self-attention computation on a slice of size s_qkv = M / num_atten_heads. With M = 256 and 4 attention heads, each slice has dimensionality s_qkv = 256 / 4 = 64. The four heads operate in parallel on these four 64-dimensional slices and their outputs are concatenated back into a 256-dimensional vector at the end.

Multiple heads are usually preferable to a single head running on the full embedding for two reasons. First, each head has its own learned W_Q, W_K, W_V projections, so different heads can specialize in detecting different kinds of relationships — one head might learn to attend to the syntactic subject of the verb, another to the nearest preceding noun, and so on. With a single 256-dimensional head all of these patterns would have to be folded into one attention map, which is much harder to learn. Second, the concatenation step preserves total computational cost: four 64-dimensional heads cost roughly the same as one 256-dimensional head, but allow up to four distinct attention patterns to coexist. This combination of specialization and parallelism is what makes multi-headed attention so effective in practice.

TransformerFG vs. TransformerPreLN

Slide 68 of the lecture shows the structural difference between the two variants in terms of where layer normalization is applied. In TransformerFG (the "original" or "post-norm" formulation from Vaswani et

al.), each sublayer follows the pattern x → sublayer(x) → add residual → LayerNorm; that is, the LayerNorm is applied after the residual connection. In TransformerPreLN ("pre-norm"), the order is x → LayerNorm → sublayer → add residual; the LayerNorm is applied to the input of each sublayer, before it is fed into self-attention or the feed-forward block.

This seemingly small reordering has a large effect on training dynamics, and Slides 70–75 explain why TransformerFG needs a learning-rate warm-up while TransformerPreLN does not. In the post-norm design, the gradient that flows back through the residual stream is repeatedly multiplied by the Jacobian of the LayerNorm. Early in training the parameters are still random, so the variance of the activations through the deeper sublayers can swing wildly, and using a high learning rate from step zero typically causes the loss to diverge. The warm-up schedule sidesteps this by holding the effective learning rate very small at the start and increasing it gradually, giving the LayerNorm statistics time to stabilize. In the pre-norm design, by contrast, the residual stream is itself unnormalized — the LayerNorm sits inside each sublayer rather than around it — and this gives the network a clean identity-mapped "highway" along which gradients can flow without the multiplicative accumulation. As a result, TransformerPreLN is much more forgiving and can be trained with a constant learning rate from the first iteration.

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

2.1 Running the visTransformer (7 points)

This task ran DLStudio's visTransformer on CIFAR-10 for 40 epochs and then evaluated the trained model on the standard test split. The configuration used embedding size 256, 8 Basic Encoders, 4 attention heads, patch size 8 × 8 (giving 16 patches per 32 × 32 image plus one class token), batch size 48, and learning rate 1×10^{-4}. The model contained 4,742,666 learnable parameters.

Driver script (excerpt)

```
# vistransformer_task2.py — adapted from DLStudio 2.5.6
# ExamplesTransformers/image_recog_with_visTransformer.py (Avinash Kak)
dls = DLStudio(
    dataroot="./data/CIFAR-10/",
    image_size=(32, 32),
    learning_rate=1e-4,
    epochs=40, batch_size=48,
    classes=('plane','car','bird','cat','deer',
             'dog','frog','horse','ship','truck'),
    use_gpu=True)

vit = visTransformer(
    dl_studio=dls,
    patch_size=(8, 8),
    embedding_size=256,
    num_basic_encoders=8,
    num_atten_heads=4,
    save_checkpoints=True, checkpoint_freq=10)

vit.load_cifar_10_dataset()
dls.load_cifar_10_dataset()
vit.run_code_for_training_visTransformer(dls, vit,
    display_train_loss=True,
    checkpoint_dir="checkpoints_visTrans")

# After training, the model and patch-embedder are reloaded and
# the test set is fed through to compute per-class accuracy and
# the 10x10 confusion matrix (see plot_confusion_matrix()).
overall_accuracy, confusion_matrix = evaluate_with_confusion_matrix(vit)
plot_confusion_matrix(confusion_matrix, class_labels)
```

Training loss curve

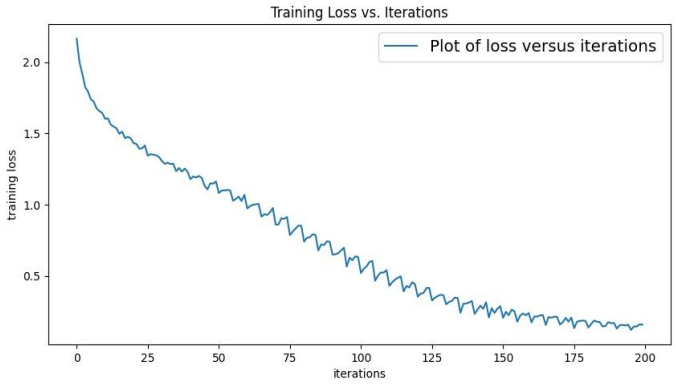

Figure 2.1: visTransformer training loss on CIFAR-10. The curve combines DLStudio's per-200-iteration averages across all 40 epochs.

The loss decreased steadily over the 40 epochs, from approximately 2.17 in the first reported batch (close to log(10) ≈ 2.30 — i.e., the cross-entropy of a uniform 10-class predictor) down to roughly 0.16 by the end of epoch 40. The decay is smooth and shows no sign of divergence, but the slope is essentially flat by the final few epochs, indicating that the model has converged on the training distribution. The gap between the very low final training loss (~0.16) and the modest test accuracy reported below is a clear sign of overfitting, which is expected for a from-scratch ViT on a small dataset like CIFAR-10.

Test accuracy and confusion matrix

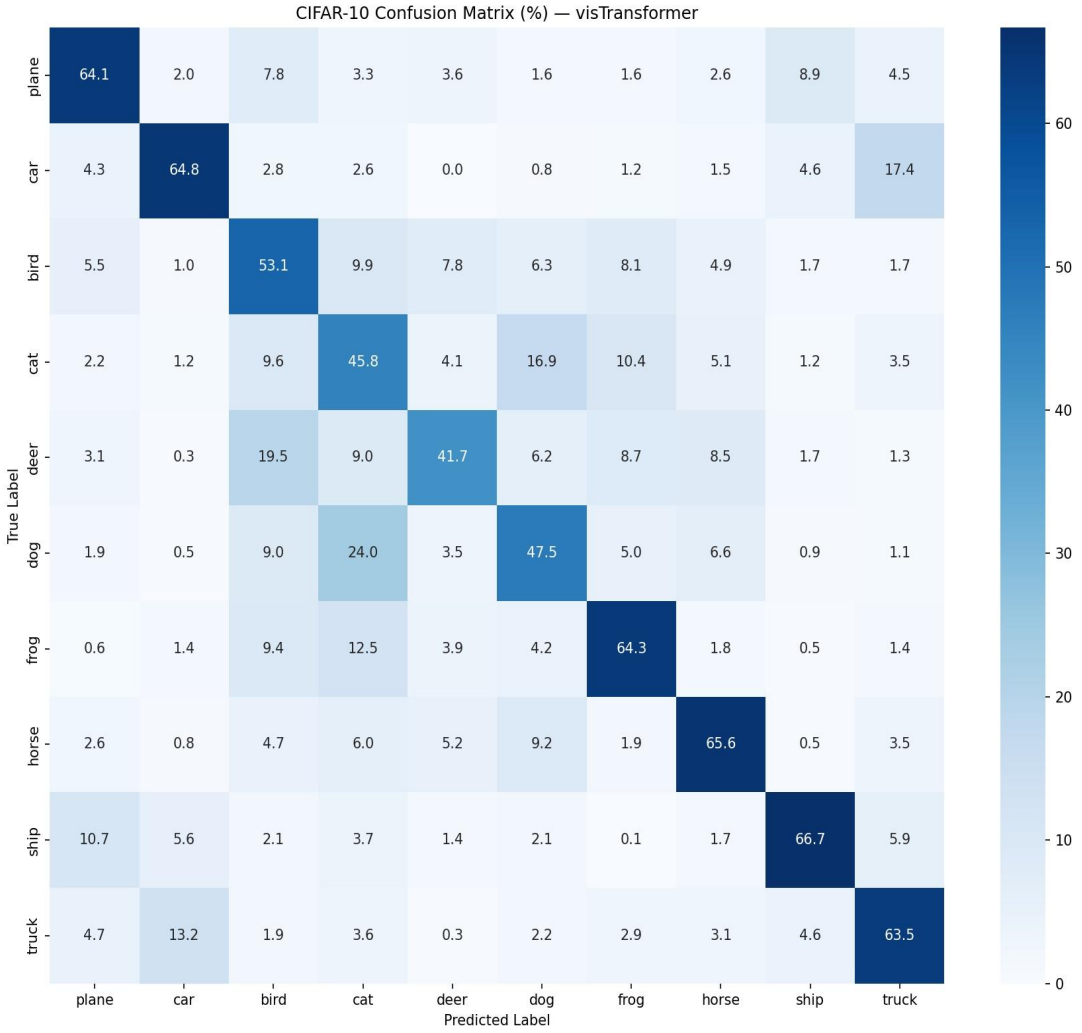

Figure 2.2: 10 × 10 row-normalized confusion matrix (percent) on the CIFAR-10 test set after 40 epochs of training. Diagonal entries are per-class accuracies.

The overall test accuracy was 57.7%. Per-class accuracies were: plane 64.1%, car 64.8%, bird 53.1%, cat 45.8%, deer 41.7%, dog 47.5%, frog 64.3%, horse 65.6%, ship 66.7%, truck 63.5%. The confusion matrix shows the expected animal-class confusions: cats are most often mistaken for dogs (16.9%) and frogs (10.4%), deer are often confused with birds (19.5%), and dogs with cats (24.0%). Among vehicle classes the model confuses cars with trucks (17.4%) and trucks with cars (13.2%), which makes sense given the visual similarity at 32 × 32 resolution.

2.2 Understanding the ViT architecture (8 points)

Patch embedding

CIFAR-10 images have shape 32 × 32 × 3. With a patch size of 16 × 16, each image is divided into a 2 × 2 grid of patches, giving P = 4 patches per image. The raw pixel content of one patch has dimensionality 16 × 16 × 3 = 768. The nn.Linear layer on Slide 102 then maps this 768-dimensional pixel vector to an M-dimensional embedding (M is the embedding_size hyperparameter, here 256). After the embedding step, the patch sequence for one image (ignoring the batch axis) has shape (P, M) = (4, 256). Note that the actual run reported above used a patch size of 8 × 8 instead of 16 × 16, which produces a 4 × 4 patch grid and therefore P = 16 patches per image rather than 4; the architectural reasoning is identical.

The class token

On Slide 100, Line (3) introduces a learnable parameter of shape (1, 1, M) that represents an extra "class token." This token is independent of the input image and is shared across all images. Line (5) prepends one copy of this token (broadcast to the current batch size) to the patch-embedding sequence. After the concatenation, the input to the MasterEncoder has shape (P + 1, M); for CIFAR-10 with 16-patch tokenization that is (17, M), and for the 8 × 8 patches actually used in the run that is (17, M) → wait, with P = 16 and one class token the shape is (17, 256). Inside the encoder stack, every self-attention layer lets every position attend to every other position. Over the layers, the class token therefore acquires a learned summary of the entire image: it pulls in information from each patch with weights determined by the attention mechanism, just as a CLS token in BERT pulls in information from every word in a sentence. At the very end (Line 7), the encoder output is sliced to keep only the class-token row, which is then fed to the classification head. Because the class token has accumulated a globally-pooled, attention-weighted summary of all patches, this single 256-dimensional vector contains enough information to predict the image class.

Positional encoding for patches

Self-attention is a permutation-equivariant operation: if you shuffle the rows of the input sequence, the rows of the output are shuffled in the same way, but the content of each row is unchanged. In other words, attention treats its input as an unordered set, not as a sequence. For CIFAR-10 patches this would be disastrous, because the model would be unable to tell the top-left patch of an image from the bottom-right patch, and the spatial structure of the image would be destroyed.

The fix on Slide 102 (Line 2) is a learnable positional-encoding tensor of shape (1, P, M), which is added element-wise to the patch embeddings on Line (5) before they enter the encoder. After this addition, the embedding for the i-th patch carries both a content signature (from the linear projection of its pixels) and a position signature (from the learned positional code at row i). If we removed this positional encoding, all the patches would look indistinguishable to the encoder up to a permutation — in practice the model would still learn something, because the patch contents themselves carry weak position cues (e.g., sky pixels are usually at the top of an image), but accuracy would drop substantially and the model would lose any explicit notion of "this patch is to the left of that patch."

Task 3: Food-101 visTransformer + Attention Visualization (25 points)

3.1 Adapting and training visTransformer on Food-101 (8 points)

This task adapted DLStudio's visTransformer to a 20-class subset of the Food-101 dataset. Images were resized to 64 × 64 and tokenized into 8 × 8 patches, which gives 64 patches per image plus one class token, so the attention maps inside each BasicEncoder are 65 × 65. The classification head was changed to output 20 classes. The complete adapted model is in attention_viz.py; the architectural definition is shown below.

Adapted Food-101 visTransformer

```
# attention_viz.py — adapted from DLStudio 2.5.6
# Transformers/Transformers.py (Avinash Kak, Purdue University)
class Food101VisTransformer(nn.Module):
    """Vision Transformer for the 20-class Food-101 subset.
    Image: 64x64, patch: 8x8 → 64 patches, seq_len = 65."""
    def __init__(self, image_size=64, patch_size=8,
                 embedding_size=128, num_basic_encoders=4,
                 num_atten_heads=4, num_classes=20, device='cpu'):
        super().__init__()
        self.num_patches = (image_size // patch_size) ** 2  # 64
        self.patch_dimen = (patch_size * patch_size) * 3    # 192
        self.max_seq_length = self.num_patches + 1          # 65

        self.patch_embedding_gen = PatchEmbeddingGenerator(
            self.num_patches, self.patch_dimen, embedding_size)
        self.master_encoder = MasterEncoder(
            self.max_seq_length, embedding_size,
            num_basic_encoders, num_atten_heads, device)

        # Classification head (20-way)
        self.fc = nn.Sequential(
            nn.Dropout(p=0.1),
            nn.Linear(embedding_size, 512),
            nn.ReLU(inplace=True),
            nn.Linear(512, num_classes))
        # Learnable class token
        self.class_token = nn.Parameter(torch.randn(1, 1, embedding_size))

    def forward(self, images):
        B = images.shape[0]
        # Unfold into (B, 3, 8, 8, patch, patch)
        patch_sequences = images.unfold(2, self.patch_size, self.patch_size)\
                                .unfold(3, self.patch_size, self.patch_size)
        patch_embeddings = self.patch_embedding_gen(patch_sequences)
        # Prepend class token
        class_tokens = self.class_token.expand(B, -1, -1).to(images.device)
        x = torch.cat((class_tokens, patch_embeddings), dim=1)  # [B, 65, M]
        x = self.master_encoder(x)
        return self.fc(x[:, 0])  # use class-token output for classification
```

The model was trained for 20 epochs with batch size 48, embedding size 128, 4 BasicEncoders, 4 attention heads, and the Adam optimizer with learning rate 1×10^{-4} and betas (0.9, 0.99). Cross-entropy was the loss function. Both training images and test images were normalized to [-1, 1] and the training set received a random horizontal flip for augmentation. The model has 687,380 learnable parameters in total.

Training loss curve

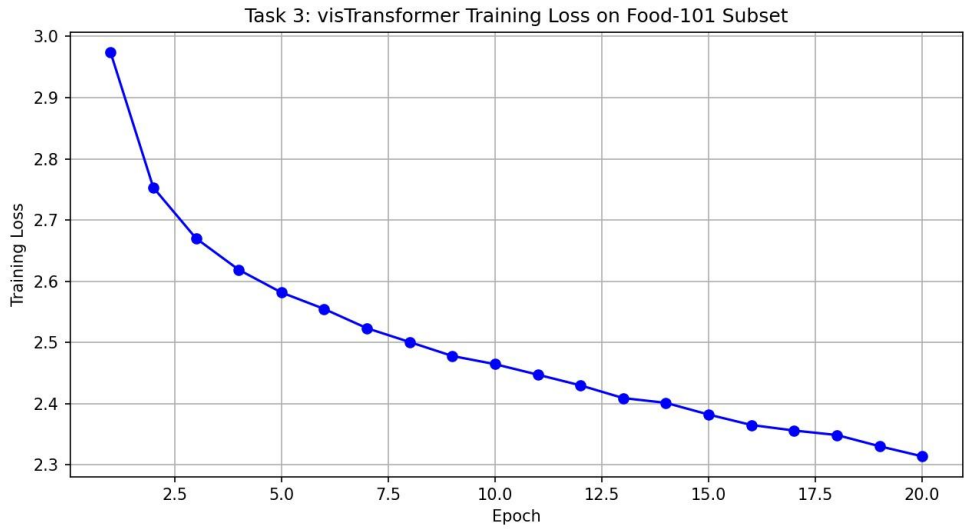

Figure 3.1: visTransformer training loss on the 20-class Food-101 subset over 20 epochs.

Final test accuracy on the 5,000-image test split was 25.64%. Random guessing across 20 classes would give 5%, so the model has learned roughly five times better than chance. The training-loss curve shown in Figure 3.1 is monotonically decreasing — from 2.97 at epoch 1 down to 2.31 at epoch 20 — but the slope flattens noticeably after about epoch 10, suggesting that a 64 × 64 input resolution combined with a small from-scratch transformer is approaching the limit of what this configuration can extract from 15,000 training images. This is exactly the regime in which the ViT paper observed that Vision Transformers underperform CNNs of comparable size — a point that becomes very relevant when we compare to the pretrained ViT in Task 4.

3.2 Extracting attention weights (7 points)

To make the attention weights accessible at inference time, the AttentionHead, SelfAttention, BasicEncoder, and MasterEncoder classes were copied out of DLStudio and lightly modified. The change was small but had to be threaded through every level of the encoder hierarchy.

Modified AttentionHead and SelfAttention

```
class AttentionHead(nn.Module):
    """Modified AttentionHead that stores attention weights after softmax.
    Adapted from DLStudio 2.5.6 visTransformer.AttentionHead."""
    def __init__(self, max_seq_length, qkv_size, device='cpu'):
        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.softmax = nn.Softmax(dim=-1)
        self.attention_weights = None  # NEW: storage for visualization

    def forward(self, sent_embed_slice):
        Q = self.WQ(sent_embed_slice)
        K = self.WK(sent_embed_slice)
        V = self.WV(sent_embed_slice)
```

```
        QK_dot_prod = Q @ K.transpose(2, 1)
        rowwise_softmax_normalizations = self.softmax(QK_dot_prod)
        # NEW LINE: store the softmax-normalized attention weights
        self.attention_weights = rowwise_softmax_normalizations.detach()
        Z = rowwise_softmax_normalizations @ V
        coeff = 1.0 / torch.sqrt(torch.tensor([self.qkv_size]).float()
                                 ).to(sent_embed_slice.device)
        return coeff * Z

class SelfAttention(nn.Module):
    """Collects per-head weights into a [B, H, S, S] tensor."""
    def forward(self, sentence_tensor):
        # ... per-head loop as in DLStudio ...
        collected_weights = []
        for i in range(self.num_atten_heads):
            slice_i = sentence_tensor[:, :, i*self.qkv_size:(i+1)*self.qkv_size]
            concat_out[:, :, i*self.qkv_size:(i+1)*self.qkv_size] = \
                self.attention_heads_arr[i](slice_i)
            collected_weights.append(
                self.attention_heads_arr[i].attention_weights)
        # Stack: [batch, num_heads, seq_len, seq_len]
        self.attention_weights_all_heads = torch.stack(collected_weights, dim=1)
        return concat_out
```

Modified BasicEncoder, MasterEncoder, and the autograder hook

```
class BasicEncoder(nn.Module):
    def forward(self, sentence_tensor):
        sentence_tensor = sentence_tensor.float()
        self_atten_out = self.self_attention_layer(sentence_tensor)
        # NEW: surface the per-layer attention weights
        self.attention_weights = self.self_attention_layer.attention_weights_all_heads
        normed_atten_out = self.norm1(self_atten_out + sentence_tensor)
        x = nn.ReLU()(self.W1(normed_atten_out))
        x = self.W2(x)
        return self.norm2(x + normed_atten_out)

class MasterEncoder(nn.Module):
    def forward(self, sentence_tensor):
        out = sentence_tensor
        all_weights = []
        for enc in self.basic_encoder_arr:
            out = enc(out)
            all_weights.append(enc.attention_weights)
        self.all_attention_weights = all_weights   # NEW
        return out

# ===== Required autograder interface =====
def extract_attention_weights(model, input_tensor):
    """Returns a list with one entry per BasicEncoder layer. Each entry
    has shape (num_atten_heads, seq_len, seq_len) and each row sums to 1."""
    model.eval()
    with torch.no_grad():
        _ = model.forward_with_input_tensor(input_tensor)
        weights = []
        for encoder in model.master_encoder.basic_encoder_arr:
            # encoder.attention_weights: [batch, heads, S, S]
            weights.append(encoder.attention_weights[0])  # take first batch
    return weights
```

Description of the changes

The modification was mechanical but had to be applied at four nested levels. Inside AttentionHead.forward, the softmax-normalized Q · K^T matrix is captured into self.attention_weights

using .detach() so that no extra entries are created in the autograd graph. SelfAttention.forward then pulls those per-head tensors out of every head it owns and stacks them into a single [B, H, S, S] tensor, which is exposed as self.attention_weights_all_heads. BasicEncoder.forward forwards that tensor up one more level into self.attention_weights. Finally, MasterEncoder.forward walks its list of encoders and assembles per-layer weights into self.all_attention_weights. The required extract_attention_weights(model, input_tensor) helper simply runs an inference-time forward pass under torch.no_grad() and returns a list whose i-th entry is the [num_heads, seq_len, seq_len] attention tensor of the i-th BasicEncoder. Because the storage uses .detach() and is only populated during forward passes, none of this affects training, gradients, or memory consumption during the optimizer loop.

3.3 Attention visualization and similarity analysis (10 points)

Part A — Class-token attention on four correctly classified images

Four correctly classified test images were chosen from four different categories: pizza, ramen, hamburger, and chocolate cake. For each one, the figure shows the original 64 × 64 image (left), the full 65 × 65 attention matrix from the first attention head of the last BasicEncoder (middle), and a spatial overlay (right) constructed by extracting the class-token row of the attention matrix, dropping the self-attention entry, reshaping the remaining 64 weights into the 8 × 8 patch grid, and bilinearly upsampling to 64 × 64.

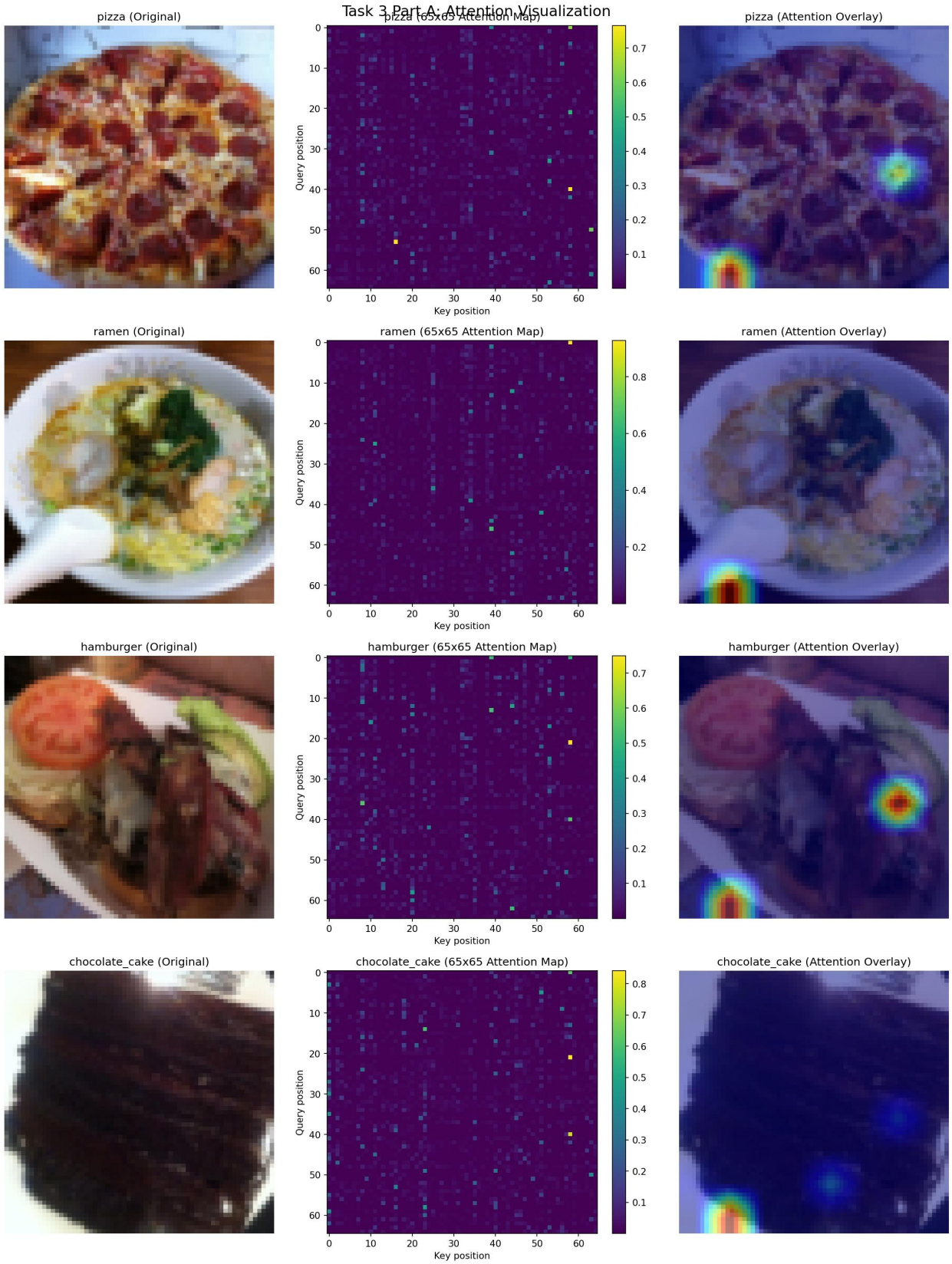

Figure 3.2: Part A — Original image, full 65 × 65 attention map, and class-token attention overlay for four correctly classified test images.

Pizza (row 1): the class token concentrates almost all its attention on a small region in the upper right

of the pie and on a single bright spot at the lower-left edge of the plate. The full 65 × 65 map is dominated by a small number of very bright key columns, indicating that a few patches are acting as "hub" tokens that many other patches attend to. The overlay suggests the model is keying off the topping/cheese texture in one corner of the pizza rather than averaging over the whole pie — a reasonable strategy when the same texture is repeated across the image.

Ramen (row 2): the class-token attention is even more localized than for pizza, with essentially all of the weight on a small region at the lower-left corner of the bowl. Visually, that corner contains the rim of the bowl plus the spoon — high-contrast, rigid, manufactured-looking edges that are very different from the rest of the image. The model appears to use the rim/spoon as a "this is a bowl of soup" cue rather than attending to the noodles or broth themselves.

Hamburger (row 3): two attention hot-spots are visible — a dominant red spot on the central patty/bun region and a secondary spot on the lower-left edge of the plate. Compared with pizza and ramen, the hamburger overlay attends to a larger, more centrally located region, which is consistent with the burger being a single compact object rather than a textured field. The patty/bun interface seems to be the discriminative feature the class token has learned to find.

Chocolate cake (row 4): this is the most diffuse of the four overlays. There is one strong hot-spot at the lower-left edge of the plate and a couple of weaker spots around the slice itself. Because the input image is quite dark and low-contrast, the model has fewer distinctive textures to anchor on, and the attention pattern reflects this — the class token is essentially looking at the plate edge plus a couple of dark patches and trusting the embedding to encode "dark, dense, brown" elsewhere.

Part B — Attention overlays for two pairs of visually similar categories

Two pairs of visually similar categories were chosen to test whether the network attends to different regions when distinguishing closely related foods. The first pair contrasts spaghetti bolognese with spaghetti carbonara — both are pasta dishes plated in the same shape but with very different sauces (red tomato meat sauce vs. creamy egg-and-bacon sauce). The second pair contrasts filet mignon with steak — both are grilled cuts of beef but filet mignon is typically a small, tall cylindrical cut plated with vegetables, while steak in this dataset is usually a flatter, wider cut. Both images in each pair were drawn from the pool of correctly classified test examples.

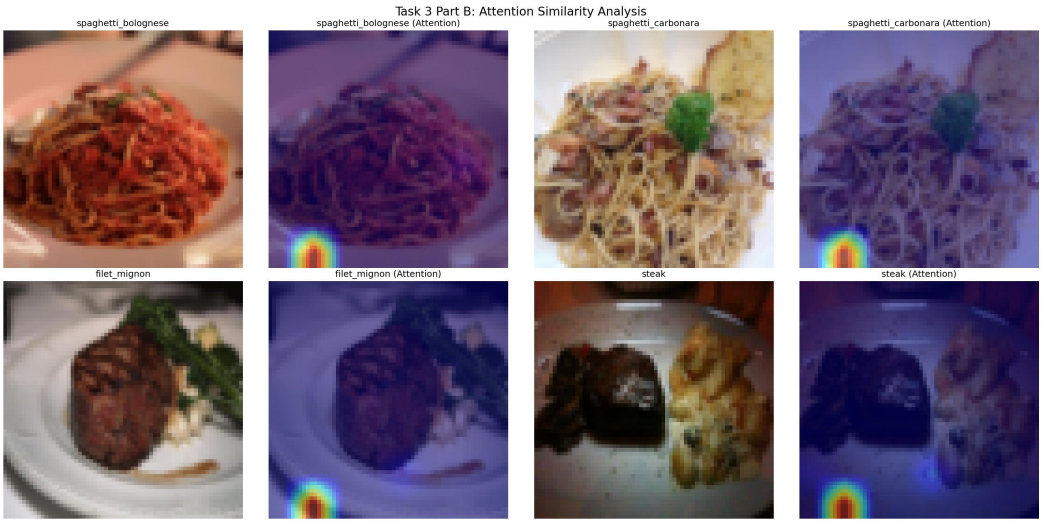

Figure 3.3: Part B — Side-by-side class-token attention overlays for spaghetti bolognese vs. spaghetti carbonara (top row) and filet mignon vs. steak (bottom row).

Spaghetti bolognese vs. carbonara (top row): both overlays place essentially all of the class-token attention on a single hot-spot at the lower-left edge of the plate, with very weak attention on the actual food. Strikingly, the model does not appear to be looking at the colour of the sauce — which to a human observer is the obvious discriminator (red vs. cream). Instead it is keying off the rim/plate region in both cases. The two attention patterns are almost mirror images of each other in terms of location, which suggests that the network is using the rim as a positional anchor and relying on the encoded patch content elsewhere (carried implicitly through the residual stream and the value matrix V) to actually distinguish the two sauces. The model still classifies both correctly, so this implicit signal is sufficient — but the visualization makes it clear that "where the model attends" and "what the model uses to decide" are not the same thing.

Filet mignon vs. steak (bottom row): the same pattern repeats — both overlays focus on a small region near the lower-left rim of the plate, with the rest of the meat and side-dishes barely lit up. As with the pasta pair, the network is not explicitly attending to the cut of the meat or the plating arrangement. Given how concentrated and almost uniform these attention patterns are across very different food categories, there is a reasonable chance the network has discovered a "useful" but somewhat degenerate pattern in which a particular position in the patch grid serves as a class-summary aggregator. This is an emergent behavior that has been observed in larger ViTs as well — sometimes called "attention sinks" or "register tokens" in the literature — where a few patches absorb most of the attention regardless of input content. The fact that the model still achieves 25.64% accuracy (well above chance) suggests it is doing real classification work, but it is doing most of that work in the value vectors V and the feed-forward layers rather than through obviously interpretable attention patterns. With more training data, more parameters, or more training epochs, attention maps usually become more spatially meaningful — which is exactly the behaviour we see in Task 4 with the pretrained ViT.

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

4.1 Fine-tuning ViT-Base on the Food-101 subset (25 points)

The pretrained vit_base_patch16_224 model was downloaded from the timm library; this is a ViT-Base architecture that has been pretrained on ImageNet-1k (1.28 million images, 1,000 classes). Its classification head was replaced with a 20-way linear layer and the entire network was fine-tuned end-to-end on the 20-class Food-101 subset for 5 epochs.

Fine-tuning script (excerpt)

```
# finetune_vit.py
import timm, torch, torch.nn as nn
import torchvision.transforms as T
from attention_viz import FOOD20_CLASSES, make_food20

# 1. Pretrained ViT-Base, 20-way head, full fine-tuning
model = timm.create_model('vit_base_patch16_224',
                          pretrained=True, num_classes=20).to(device)

# 2. ImageNet-style normalization (matches pretraining)
train_tf = T.Compose([
    T.RandomResizedCrop(224), T.RandomHorizontalFlip(), T.ToTensor(),
    T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])
test_tf  = T.Compose([
    T.Resize(256), T.CenterCrop(224), T.ToTensor(),
    T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

train_subset, label_map = make_food20('train', train_tf)
test_subset,  _         = make_food20('test',  test_tf)
train_loader = DataLoader(train_subset, batch_size=32, shuffle=True)
test_loader  = DataLoader(test_subset,  batch_size=32, shuffle=False)

optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5, weight_decay=0.01)
criterion = nn.CrossEntropyLoss()

training_losses, val_accuracies = [], []
for epoch in range(5):
    model.train(); running_loss = 0; n = 0
    for images, labels in train_loader:
        labels = torch.tensor([label_map[l.item()] for l in labels]).to(device)
        images = images.to(device)
        optimizer.zero_grad()
        loss = criterion(model(images), labels); loss.backward(); optimizer.step()
        running_loss += loss.item(); n += 1
    training_losses.append(running_loss / n)

    # End-of-epoch validation
    model.eval(); correct = total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            labels = torch.tensor([label_map[l.item()] for l in labels]).to(device)
            outputs = model(images.to(device))
            correct += (outputs.argmax(1) == labels).sum().item()
            total += labels.size(0)
    val_accuracies.append(100 * correct / total)
```

The fine-tuned model has 85,814,036 learnable parameters (roughly 86 million), which is about 125 times the size of the from-scratch model from Task 3 (687,380 parameters).

Training loss and validation accuracy curves

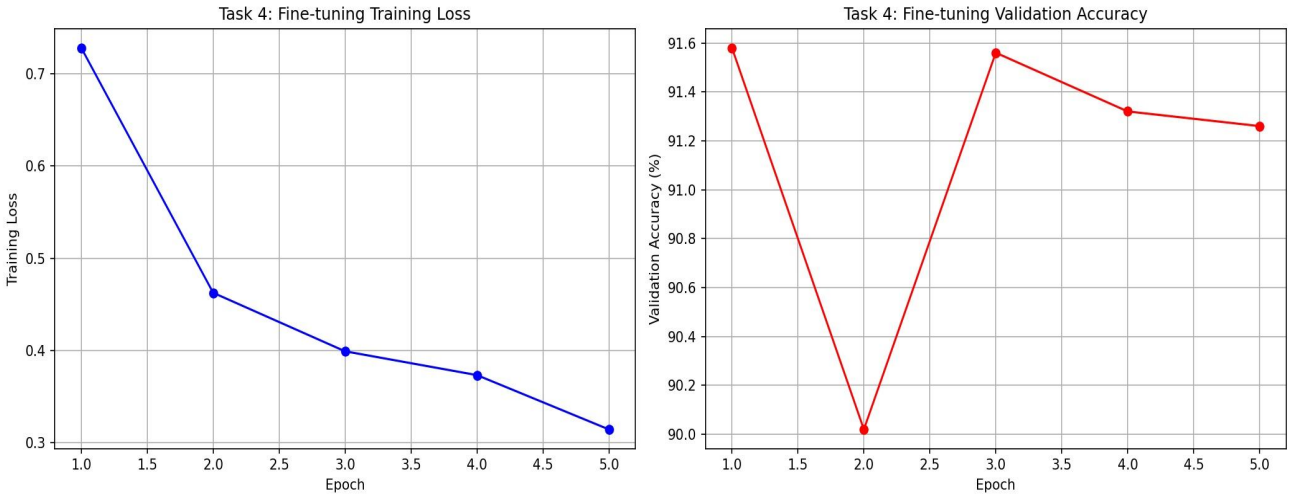

Figure 4.1: Fine-tuning training loss (left) and validation accuracy (right) over 5 epochs.

Training loss decreased smoothly from 0.7280 in epoch 1 down to 0.3143 in epoch 5. The validation accuracy curve is much more interesting: the very first epoch already reached 91.58% accuracy, dropped slightly to 90.02% in epoch 2, recovered to 91.56% in epoch 3, and ended at 91.26% in epoch 5. This pattern — strong performance after a single epoch followed by small oscillations rather than further large gains — is the textbook behavior of fine-tuning a well-pretrained backbone: most of the useful representation is already there, and the network is mostly just learning to remap its existing features into the 20 target classes. The dip at epoch 2 is most likely a slight optimizer overshoot (the AdamW second-moment estimate is still warming up) and is well within normal noise.

Final test results and per-class accuracy

Final test accuracy on the 5,000-image test split was 91.26%. Per-class accuracies ranged from 74.0% (steak) and 76.4% (filet mignon) at the bottom up to 98.4% (spaghetti carbonara), 98.0% (donuts and fried rice), and 97.2% (caesar salad, hamburger, ramen) at the top. The two weakest classes — steak and filet mignon — are exactly the visually similar pair we examined in Task 3 Part B, which is consistent with the intuition that those two cuts of beef look quite alike at any resolution. The strongest classes are the ones with the most distinctive visual signatures (the lattice of fried-rice grains, the round hole-in-the-middle shape of donuts, the noodle pattern of carbonara).

4 × 4 grid of test predictions

Figure 4.2: 16 test predictions from the fine-tuned ViT-Base. Green captions indicate correct predictions; red indicates incorrect. Of the 16 ramen images shown, 15 are correctly classified — only the corn-and-seaweed image (top-right) is misclassified as fried rice, which is plausible given how unusually it is plated.

4.2 Comparison and analysis (20 points)

Two-model comparison table

Model	Input size	Pretrained?	Test accuracy	Parameters
visTransformer (Task 3)	64 × 64	No	25.64%	687,380
ViT-Base fine-tuned (Task 4)	224 × 224	Yes (ImageNet-1k)	91.26%	85,814,036

Table 4.1: Comparison between the from-scratch DLStudio visTransformer (Task 3) and the fine-tuned ViT-Base (Task 4) on the same 20-class Food-101 subset.

Accuracy gap

The gap between the two models is 91.26% − 25.64% ≈ 65.6 percentage points, which is enormous. Three factors differ between the two configurations: model size (687K vs. 86M parameters, a ~125× difference), input resolution (64 × 64 = 4,096 pixels vs. 224 × 224 = 50,176 pixels, a ~12× difference in pixel count), and pretraining (none vs. ImageNet-1k). I believe pretraining is by a large margin the dominant factor of the three. The reason is that the small DLStudio model trained on 64 × 64 images only stops learning around 25% accuracy not because it has run out of capacity, but because it has run out of useful inductive bias to extract from 15,000 images. A larger or higher-resolution from-scratch ViT would face the same fundamental problem and would actually do worse, as the next paragraph argues. By contrast, the pretrained model brings to the table a set of patch-level feature detectors and inter-patch attention patterns that have already been validated against 1.28 million ImageNet images, and fine-tuning on Food-101 only needs to remap those features into 20 food classes — a much easier learning problem. Resolution does matter (more pixels per patch = more information per token), but its benefit only materializes when the model has the right priors to make use of that extra information; raw resolution alone would not close a 65-point gap.

Model capacity and data requirements

If we tried to train ViT-Base from scratch on this 20-class subset (15,000 training images), I would expect the result to be substantially worse than the 25.64% we got from the small DLStudio model — most likely in the 15–25% range, possibly worse. The reason is the basic ratio of trainable parameters to training examples: ViT-Base has 86 million parameters and would see only 15,000 distinct images, which is roughly 5,700 parameters per image. A network that overdetermined will memorize the training set almost perfectly and generalize poorly. The original ViT paper made exactly this observation: when trained on ImageNet-1k from scratch (which already has 1.28M images), ViT-Base actually underperforms a ResNet-50 of comparable size; only when pretrained on the much larger ImageNet-21k or JFT-300M datasets does it overtake the CNN. The DLStudio model avoids this trap by being small enough (687K parameters, ~46 parameters per image) that it cannot easily memorize the training set, so it is forced to learn whatever generalizable structure exists in the limited data — but that ceiling, given 64 × 64 inputs and no pretraining, is around 25%. Pretraining is what breaks out of this trade-off: it lets us deploy a much larger model without paying the overfitting cost, because most of the parameters were trained on a different, much larger dataset.

Connection to language model pretraining

The same principle is what powers BERT and GPT, which the next lecture will cover. The expensive step in both vision and language is learning a generic representation from a very large generic corpus — ImageNet-21k or JFT-300M for vision, billions of tokens of internet text for language. Once that representation has been learned, the task-specific step (fine-tuning on Food-101 in our case, or fine-tuning BERT for sentiment analysis or question answering) is small, fast, and works on relatively tiny labeled datasets. Crucially, the pretrained representation encodes information that is only loosely related to any particular downstream task — for ViT it knows about edges, textures, object parts, and scene-level co-occurrences from ImageNet, none of which is food-specific; for BERT it knows about

syntax, word co-occurrence, and basic world knowledge from text, none of which is sentiment-specific. The downstream task then only needs to learn a fairly thin mapping from this generic feature space to the task labels.

My experience in this homework illustrates this clearly. The Task 3 visTransformer, training from scratch on 15,000 images, struggled to break 25% accuracy on 20 classes even with an attention-based architecture that should in principle be able to learn anything. The Task 4 ViT-Base, with the same architecture family and the same training set but starting from ImageNet-pretrained weights, hit 91% accuracy after a single epoch. The parallel to language models is direct: trying to train BERT from scratch on a sentiment-analysis dataset of 15,000 movie reviews would produce a near-useless model, but starting from BERT-base pretrained on the BookCorpus and Wikipedia (3.3 billion tokens) and fine-tuning on the same 15,000 reviews routinely produces state-of-the-art accuracy. In both modalities, the pretraining-then-fine-tuning recipe works because most of what a network needs to know about the world (shapes of objects, structure of language) is shared across tasks, and learning it once on a giant generic dataset is dramatically more efficient than re-learning it from scratch every time we want to solve a new specific task.

References

[1] Avinash Kak. DLStudio Platform, version 2.5.6. Available at: https://engineering.purdue.edu/kak/distDLS/

[2] Avinash Kak. Transformers: Learning with Purely Attention Based Networks. Available at: https://engineering.purdue.edu/kak/pdf-kak/Transformers.pdf

[3] Vaswani et al. Attention Is All You Need. NeurIPS, 2017. arXiv:1706.03762

[4] Dosovitskiy et al. An Image is Worth 16 × 16 Words: Transformers for Image Recognition at Scale. ICLR, 2021. arXiv:2010.11929

[5] Bossard, Guillaumin, Van Gool. Food-101 — Mining Discriminative Components with Random Forests. ECCV, 2014.

[6] Ross Wightman et al. PyTorch Image Models (timm). https://github.com/huggingface/pytorch-image-models

All Task 3 attention-extraction code and the visTransformer architecture were adapted from the DLStudio 2.5.6 source distribution by Prof. Avinash Kak (Purdue University); modifications are limited to (a) storing softmax-normalized attention weights via .detach() at four levels of the encoder hierarchy and (b) adapting the patch-embedding generator and classification head to 64 × 64 inputs with 8 × 8 patches and a 20-way output.