Multimodal Contextual Analysis for Cross Domain Few Shot Learning

IAS-NASI-INSA
SUMMER RESEARCH FELLOWSHIP PROGRAMME 2023

A Bhavana (ENGS1776)
Guide : Dr.Soma Biswas
IACV Lab, IISc Bangalore

1 Introduction

Neural networks require massive amounts of labelled data to carry out computer vision tasks such as image classification, segmentation and object detection. The ability of a model to generalize is closely connected to the dataset used for training. Practically, collecting and annotating such large amounts of data is difficult and expensive.

For example, obtaining sufficient data in the medical domain for diagnosis of certain rare diseases or obtaining samples of certain categories like crash debris in a domain like satellite images is virtually impossible. Hence learning to carry out computer vision tasks using limited labelled data is essential to machines. Humans are quick to recognize objects in different settings after seeing just a handful of examples and generalize fairly well to recognize new classes. However even the performance of state-of-art neural networks fail in such low-data regimes.

This project seeks to analyze the possibility of applying emerging prompt learning architecture to carry out cross domain few shot learning by adding contextual and multimodal dimensions.

2 Few Shot Learning (FSL)

The problem of learning to classify using limited labelled samples is the essence of few shot learning. It is a transfer learning technique designed to learn novel classes from limited annotated labels.

The two main assumptions to ensure its successful application are that the classes between the train and finetune process must be distinct that is $C_a \cap C_b = \emptyset$. The annotation labels for each class in the finetune process must be limited. Consider two datasets D_a, D_b where classes $C_a \in D_a$ have sufficient labels and $C_b \in D_b$ only have limited labels. The objective of few shot learning task is to apply a few annotated labels to finetune novel classes of datasets D_b.

Some of the approaches adopted for FSL are

- Meta Learning
- Generative and Adversarial methods
- Semi-supervised Learning

- Transfer Learning

In meta learning or 'learning to learn', the objective of the models are to rapidly learn parameters to adapt using limited samples. Generative and augmentation methods look into creating samples from the limited available data for training the model. In the semi-supervised setting the data includes additional unlabelled data to form support sets that are assumed to have the similar distributions as the target classes. Transfer learning uses the features learned on the base classes and finetunes it on the novel classes.

3 Cross Domain Few Shot Learning (CD-FSL)

. Cross Domain Few Shot Learning is an extension of few shot learning which assumes significant domain gap between source and target domains. The CD-FSL task is usually approached in a two-fold manner :

- initially learning a universal feature extractor using the training samples from different datasets.
- adapting the model to novel tasks the new domains which have limited labelled examples.

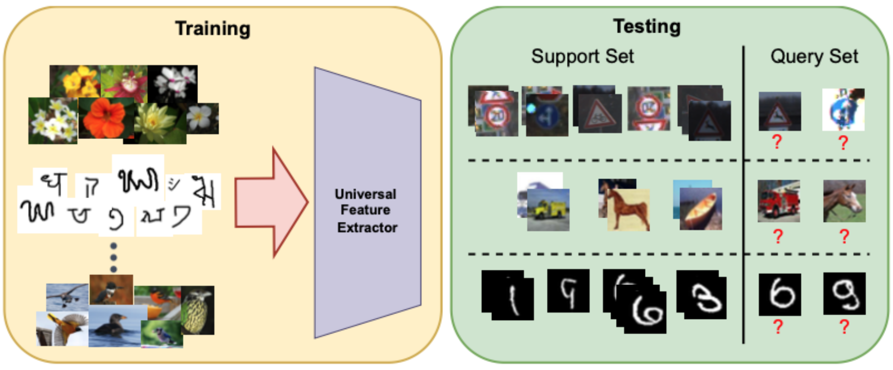


Figure 1: Cross Domain Few Shot Learning

The CD-FSL problem is described as a N-way K-shot where N represents the number of classes and K represents the number of samples per class. Each task $T = (S, Q)$ consists of a support set S and a query set Q. The model is expected to learn from the few samples in the support set $S = (x_i^S, y^Si), i = 1...n_S$ and is tested on a query set $Q = x_i^q, i = 1...n_Q$ which is assumed to contain the same classes as the corresponding support set where (x_i^S, y_i^S) represent the support set sample and label respectively, x_i^q represents the query set sample and n_S and n_Q represent the number of samples in the support set and query set respectively. the universal feature extractor F for a sample x_i^S is denoted as $z_i^S = F(x^S i)$. In the CD-FSL setting comprise of M_{train} and M_{test} datasets such that

$$\begin{aligned} D^{train} &= D_1 \cup D_2 \cup ... \cup D_{M_{train}} \\ D^{test} &= D_1^{'} \cup D_2^{'} \cup ... \cup D_{M_{test}}^{'} \end{aligned}$$

The aim is to train a universal feature extractor F on D^{train} and adapt this to carry out novel few shot tasks T from D^{test} which has novel classes and domains.

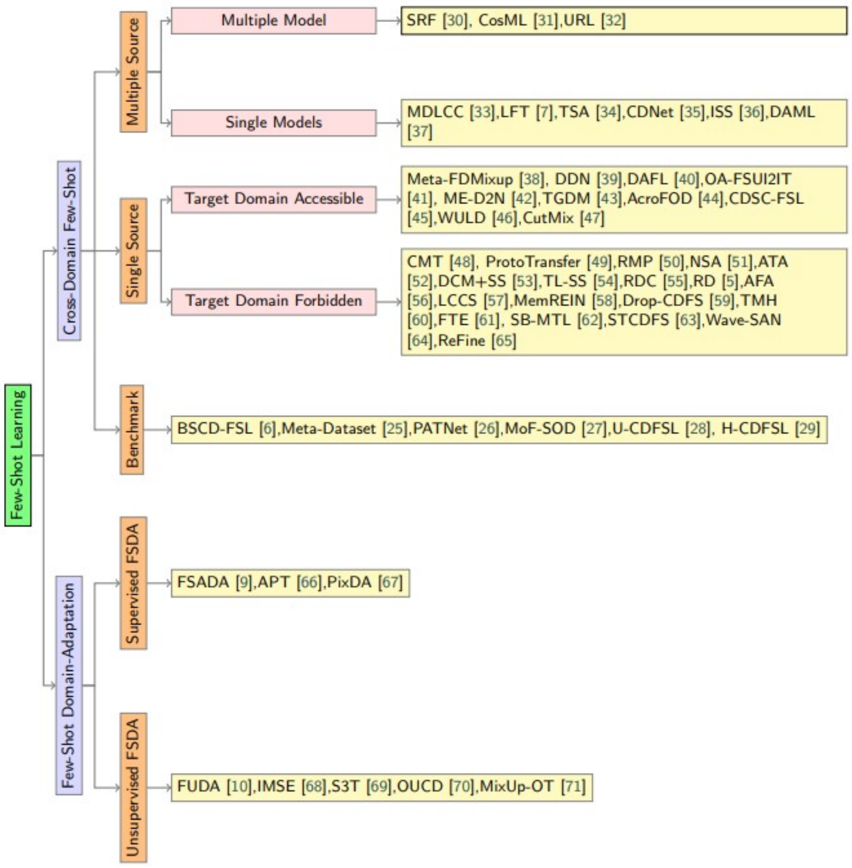

Few-Shot Learning
Cross-Domain Few-Shot
Multiple Source
Multiple Model
SRF [30], CosML [31],URL [32]
Single Models
MDLCC [33],LFT [7],TSA [34],CDNet [35],ISS [36],DAML [37]
Single Source
Target Domain Accessible
Meta-FDMixup [38], DDN [39],DAFL [40],OA-FSUI2IT [41], ME-D2N [42],TGDM [43],AcroFOD [44],CDSC-FSL [45],WULD [46],CutMix [47]
Target Domain Forbidden
CMT [48], ProtoTransfer [49],RMP [50],NSA [51],ATA [52],DCM+SS [53],TL-SS [54],RDC [55],RD [5],AFA [56],LCCS [57],MemREIN [58],Drop-CDFS [59],TMH [60],FTE [61], SB-MTL [62],STCDFS [63],Wave-SAN [64],ReFine [65]
Benchmark
BSCD-FSL [6],Meta-Dataset [25],PATNet [26],MoF-SOD [27],U-CDFSL [28], H-CDFSL [29]
Few-Shot Domain-Adaptation
Supervised FSDA
FSADA [9],APT [66],PixDA [67]
Unsupervised FSDA
FUDA [10],IMSE [68],S3T [69],OUCD [70],MixUp-OT [71]

Figure 2: Classification of existing models for CD-FSL

Recent approaches for this task focus on task specific model adaptation in which models employ additional task specific parameters and finetune this model using the support set samples and a nearest centroid classifier. The Meta Dataset is the benchmark that is used for the evaluation of CD-FSL task.

4 Vision Transformer (ViT)

Vision transformers models have consistently obtained state-of-art results in computer vision tasks such as object detection. In contrast to standard CNNs, which cannot model long range pixel dependencies attributed to its receptive field while Vision Transformers use global attention.

The transformer takes a 1-D sequence of token embedding as input and reshapes a 2-D image $x \in R^{H \times W \times C}$ into a sequence of flattened 2-D patches $x_p \in R^{N \times (P^2.C)}$ here (H, W) is the resolution of the original image,C is the number of channels (P, P) is the resolution of each image patch,and $N = HW/P^2$ is the resulting number of patches, effectively the input sequence length.

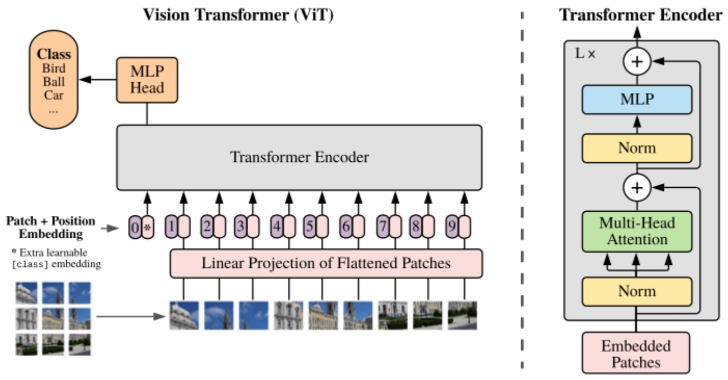


Figure 3: Vision Transformer Architecture

A sequence of embedded image sections is created by converting flattened patches into a space of dimensionality D which is done through a linear projection E that can be trained. This sequence of embedded patches is then enriched with a class-specific embedding labeled as x_{class}, which signifies the ultimate classification result y. The patch embeddings are then augmented with positional embedding E_{pos} which introduces positional information to the input that is learnt during the training. The sequence of embedded vectors are denoted as

$$z_0 = [x_{class}; x_p^1 ... x_p^N] + E_{pos}$$

To perform classification z_0 is fed into the input of the Transformer encoder comprising of L identical layers. The value of x_{class} at the L^{th} layer of the encoder output is fed into the classification head that is implemented by a multi-layer perceptron (MLP). The MLP head attached to the output of the encoder receives the value of the learnable class embedding, to generate a classification output based on its state.

```
import torch
import torch.nn as nn
import math

class ViTBlock(nn.Module):
    def __init__(self, embedding_dim, num_heads, mlp_ratio=4,
       dropout_rate=0.1):
        super().__init__()
        self.attention =
           nn.MultiheadAttention(embed_dim=embedding_dim,
           num_heads=num_heads, dropout=dropout_rate)
        self.norm1 = nn.LayerNorm(embedding_dim)
        self.mlp = nn.Sequential(
            nn.Linear(embedding_dim, mlp_ratio * embedding_dim),
            nn.GELU(),
            nn.Linear(mlp_ratio * embedding_dim, embedding_dim),
            nn.Dropout(dropout_rate)
        )
        self.norm2 = nn.LayerNorm(embedding_dim)

    def forward(self, x):
        attn_output, _ , _ = self.attention(x, x, x)
        x = x + attn_output
        x = self.norm1(x)
        mlp_output = self.mlp(x)
        x = x + mlp_output
        x = self.norm2(x)
        return x

class VisionTransformer(nn.Module):
    def __init__(self, input_resolution, patch_size,
       num_classes, embedding_dim, num_heads, num_layers,
       mlp_ratio=4, dropout_rate=0.1):
        super().__init__()
        self.input_resolution = input_resolution
        self.embedding_dim = embedding_dim
        self.num_classes = num_classes
        self.patch_size = patch_size
        self.num_layers = num_layers

        num_patches = (input_resolution // patch_size) ** 2
        patch_dim = 3 * patch_size ** 2

        self.patch_embedding = nn.Conv2d(in_channels=3,
           out_channels=embedding_dim, kernel_size=patch_size,
           stride=patch_size)
        self.positional_embedding = nn.Parameter(torch.zeros(1,
           num_patches + 1, embedding_dim))
        self.cls_token = nn.Parameter(torch.zeros(1, 1,
           embedding_dim))
        self.dropout = nn.Dropout(dropout_rate)

        self.transformer_blocks = nn.ModuleList([
            ViTBlock(embedding_dim, num_heads, mlp_ratio,
               dropout_rate) for _ in range(num_layers)
        ])

        self.fc_head = nn.Linear(embedding_dim, num_classes)

    def forward(self, x):
        x = self.patch_embedding(x)
        x = x.flatten(2).permute(2, 0, 1)
        cls_tokens = self.cls_token.expand(x.shape[1], -1, -1)

        cls_tokens = cls_tokens.expand(-1, x.shape[1], -1)

        num_patches = x.size(1)
        positional_embeddings = self.positional_embedding[:,
           :num_patches, :]

        positional_embeddings =
           positional_embeddings.unsqueeze(0).expand(x.shape[0],
           -1, -1, -1)

        x = torch.cat((cls_tokens, x), dim=0)
        x = x.cuda()
        positional_embeddings = positional_embeddings.cuda()

        x = x + positional_embeddings
        x = self.dropout(x)


        for transformer_block in self.transformer_blocks:
            x = transformer_block(x)

        cls_token = x[0]
        x = self.fc_head(cls_token)
        return x
```

5 Vision Language Models and Prompt Learning

The fusion of vision and language demonstrates remarkable potential in learning generic visual representation and few-shot learning in a variety of downstream tasks. The emerging prompt based learning introduces a novel paradigm where the model is guided by queries during training.

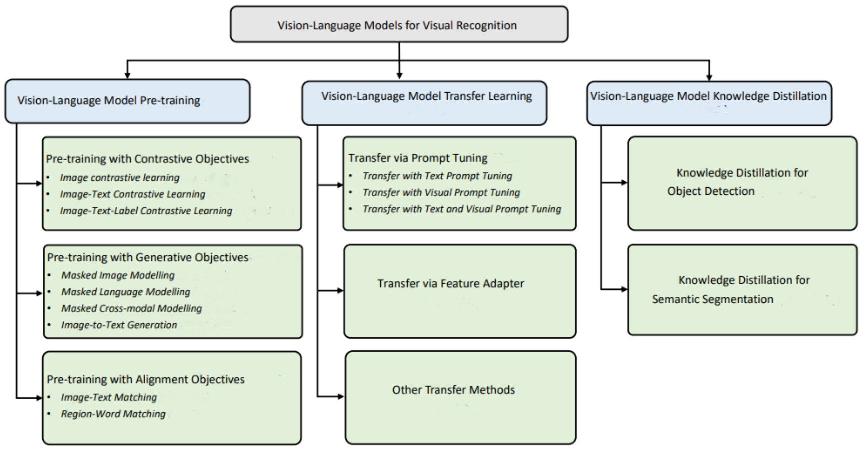

Vision-Language Models for Visual Recognition
Vision-Language Model Pre-training
Vision-Language Model Transfer Learning
Vision-Language Model Knowledge Distillation
Pre-training with Contrastive Objectives
Image contrastive learning
Image-Text Contrastive Learning
Image-Text-Label Contrastive Learning
Pre-training with Generative Objectives
Masked Image Modelling
Masked Language Modelling
Masked Cross-modal Modelling
Image-to-Text Generation
Pre-training with Alignment Objectives
Image-Text Matching
Region-Word Matching
Transfer via Prompt Tuning
Transfer with Text Prompt Tuning
Transfer with Visual Prompt Tuning
Transfer with Text and Visual Prompt Tuning
Transfer via Feature Adapter
Other Transfer Methods
Knowledge Distillation for Object Detection
Knowledge Distillation for Semantic Segmentation

Figure 4: Types of Vision Language models for visual recognition

5.1 Multimodal Analysis

The common approaches used for multimodal learning consisted of metric learning, multilabel classification and n-gram language learning. In contemporary vision-language models the convergence of both the modalities is achieved through the joint training of two encoders. The recent achievements in vision-language models owe much to advancements in

- Transformer Architectures discussed earlier
- Contrastive Representation Learning
- Expansive training datasets

5.2 Contextual Analysis

Prompt learning involves encoding contextual information using trainable vectors. This incorporation of task-specific context leads to notable enhancements, enabling effective adaptation to various tasks through prompts. Consequently, this approach facilitates seamless knowledge transfer to different downstream tasks without the need for task-specific training data.

6 CLIP

Contrastive Language–Image Pre-training (CLIP) is a revolutionary neural network architecture builds on a large body of work on zero-shot transfer and multimodal learning. CLIP employs a Convulution neural network as its image encoder to take raw images and process it through several convulution layers followed by pooling and fully connected layers. The resulting output of the image encoder is an embedding vector that represents the content of an image.

The transformer based text encoder takes a sequences of text such as captions as input and processes them through multiple layers of self-attention and feedforward neural networks. The output is an embedding vector that captures the meaning of the input text.

The core idea of CLIP is to bring images and text closer together in a latent space. This is done through a contrastive learning objective. During training, CLIP pairs each image with a relevant textual description and ensures that their embeddings are closer to each other in the latent space, while pushing the embeddings of different images and text pairs apart.

CLIP is pretrained on a massive dataset containing millions of (image, text)

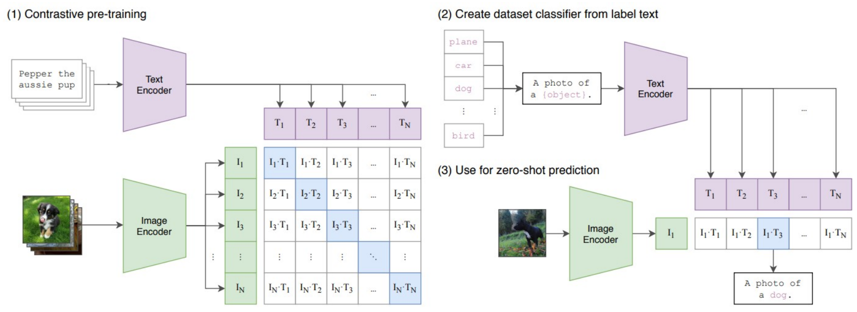

(1) Contrastive pre-training
Pepper the aussie pup
Text Encoder
Image Encoder
T_1 T_2 T_3 ... T_N
I_1 I_2 I_3 ... I_N
$I_1 \cdot T_1$ ... $I_N \cdot T_N$
(2) Create dataset classifier from label text
plane
car
dog
bird
A photo of a {object}.
Text Encoder
(3) Use for zero-shot prediction
Image Encoder
A photo of a dog.

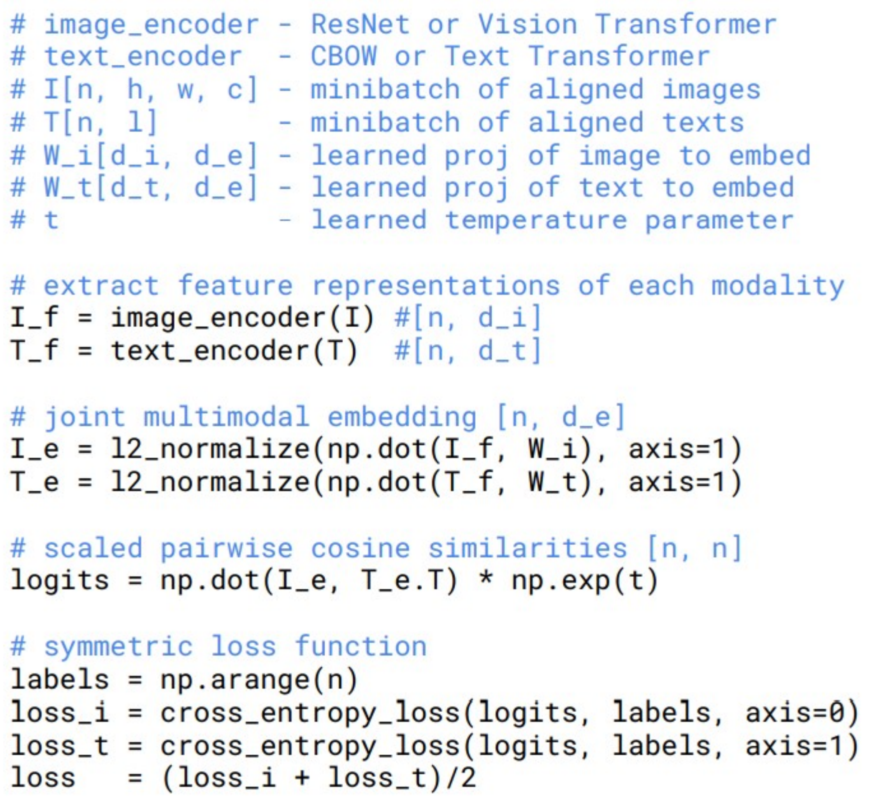
Figure 5: CLIP Architecture

pairs. These pairs are collected from the internet. This diverse dataset allows CLIP to learn to understand a wide range of images and their textual context.

Let x be image features generated by the image encoder and $w_{i}{}_{i=1}^{K}$ a set of weight vectors produced by the text encoder, each representing a category assuming there are K categories in total. In particular, each w_i is derived from a prompt, such as "a photo of a class" where the "class" token is filled with the ith class name then the probability of the prediction is given as

$$p(y|x) = \frac{\exp\left(\frac{\text{sim}(x,w_y)}{\tau}\right)}{\sum_{i=1}^{K} \frac{\text{sim}(x,w_y)}{\tau}}$$

Pretrained CLIP can perform a variety of tasks, including zero-shot image classification. To classify an image into a specific class, CLIP uses the text encoder to generate an embedding from a textual description of the category. The image's embedding is then compared to the category's embedding, and the prediction is based on their similarity.



```
# image_encoder - ResNet or Vision Transformer
# text_encoder  - CBOW or Text Transformer
# I[n, h, w, c] - minibatch of aligned images
# T[n, l]       - minibatch of aligned texts
# W_i[d_i, d_e] - learned proj of image to embed
# W_t[d_t, d_e] - learned proj of text to embed
# t             - learned temperature parameter

# extract feature representations of each modality
I_f = image_encoder(I) #[n, d_i]
T_f = text_encoder(T)  #[n, d_t]

# joint multimodal embedding [n, d_e]
I_e = l2_normalize(np.dot(I_f, W_i), axis=1)
T_e = l2_normalize(np.dot(T_f, W_t), axis=1)

# scaled pairwise cosine similarities [n, n]
logits = np.dot(I_e, T_e.T) * np.exp(t)

# symmetric loss function
labels = np.arange(n)
loss_i = cross_entropy_loss(logits, labels, axis=0)
loss_t = cross_entropy_loss(logits, labels, axis=1)
loss   = (loss_i + loss_t)/2
```

Figure 6: Pseudocode for the core of an implementation of CLIP

One of the strengths of CLIP is its flexibility. It can perform various tasks without extensive fine-tuning because its joint image-text embeddings capture a rich understanding of both modalities. This makes it versatile for tasks ranging from image classification to object detection to text generation.

7 Co-op, Co co-op and MaPLe

While models like CLIP have shown great potential in learning representations that are transferable across a wide range of downstream tasks, a major challenge for deploying them practically is prompt engineering that requires domain expertise and is time-consuming, a significant amount of time is spent on words tuning since a slight change in wording could have a huge impact on performance.

Contextual Optimization (Co-op) model was introduced with a simple approach to automate prompt engineering on such pretrained vision language models. Co-Op models the words surrounding a prompt using adaptable vectors, which can start as random values or pretrained word embeddings.

The framework offers two variations catering to different types of tasks: one involves unified context, sharing the same context across all classes and suitable for most categories; the other relies on class-specific context, which develops distinct context tokens for each class and is more appropriate for certain fine-grained categories. The core pretrained parameters remain unchanged during

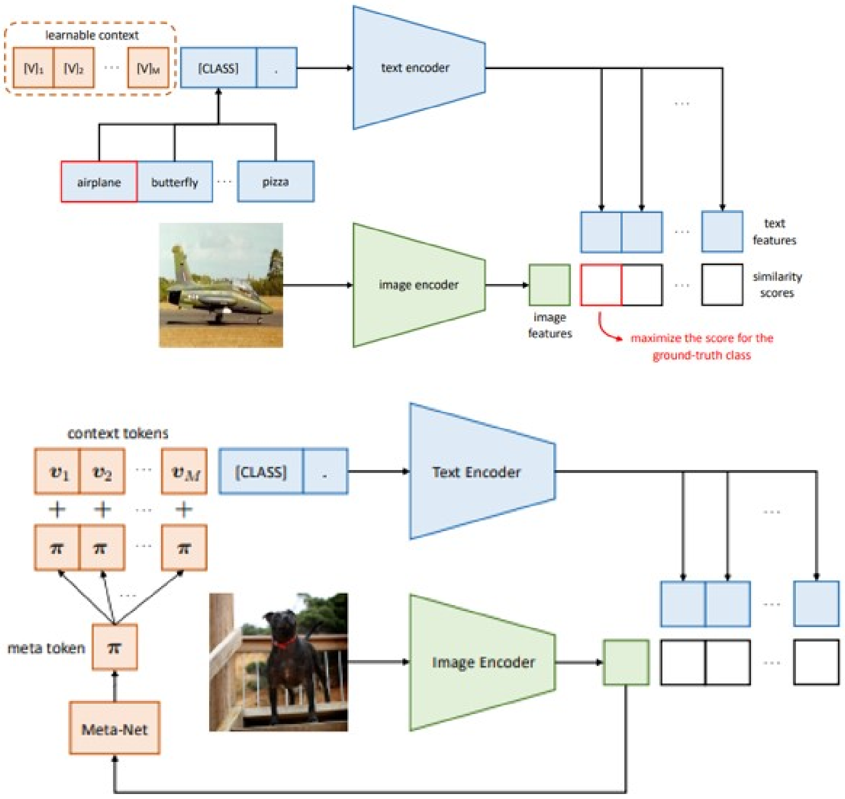


Figure 7: Co-op and Co co-op Architecture

the training process. The gradients are propagated through the text encoder, effectively extracting valuable knowledge from the parameters to facilitate the learning of context tailored to the task. However in Co-op the learned context is not generalizable to wider unseen classes within the same task.

To address this **Conditional Context Optimization** (Co co-op) was introduced. It introduces instance conditional context as a means to achieve better generalization. This shift in approach redirects the focus from specific class sets aimed at mitigating overfitting toward individual input instances. Although its performance comparatively drops in the base classes, since Co co-op optimizes for each instance in order to gain more generalization over an entire task. This further showcases the potential to transfer beyond a single dataset, that poses a much more challenging task since the fundamentals can be totally changed across different datasets.

These models however are confined to adjusting one branch of CLIP (either language or vision), which can limit the dynamic alignment of both representation spaces in response to the task at hand.

Multi-Modal Prompt Learning (MaPLe) simultaneously adapting both the vision and language branches. This approach enhances the alignment between the visual and linguistic representations.

It tightly couples the vision-language prompts to ensure a strong and mutual relationship between the two modalities. This synergy promotes a more holistic understanding of the data, that leads to improved performance.It further discourages the learning of separate solutions for each modality, favoring joint representations that capture the rich interactions between visual and linguistic information. This can prevent the model from relying solely on one modality, leading to more balanced and robust representations. It uses a stage-wise feature relationship modeling by learning distinct prompts across different early stages of the model. This allows the model to progressively capture complex context dependencies between the modalities, leading to a richer understanding of the data.

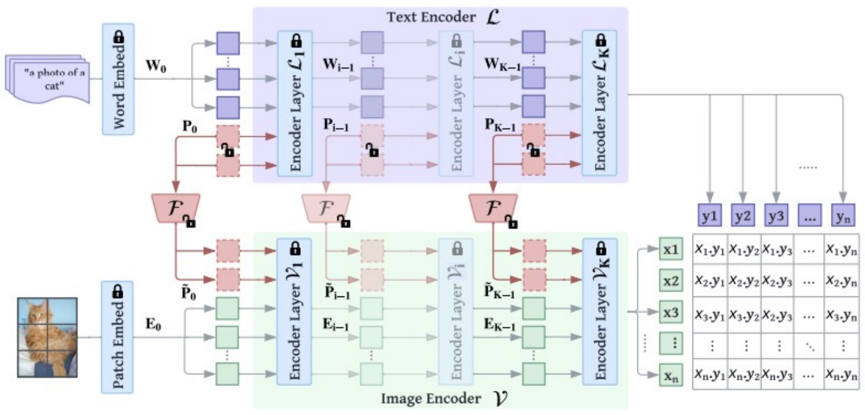

Text Encoder $\mathcal{L}$
"a photo of a cat"
Word Embed
W_0
P_0
Encoder Layer $\mathcal{L}_1$
W_{i-1}
P_{i-1}
Encoder Layer $\mathcal{L}_i$
W_{K-1}
P_{K-1}
Encoder Layer $\mathcal{L}_K$
$\mathcal{F}$
y1 y2 y3 ... yn
x1 x2 x3 ⋮ xn
$x_1 \cdot y_1$ $x_1 \cdot y_2$ $x_1 \cdot y_3$ ... $x_1 \cdot y_n$
$x_n \cdot y_1$ $x_n \cdot y_2$ $x_n \cdot y_3$ ... $x_n \cdot y_n$
Patch Embed
E_0
$\tilde{P}_0$
Encoder Layer $\mathcal{V}_1$
$\tilde{P}_{i-1}$
E_{i-1}
Encoder Layer $\mathcal{V}_i$
$\tilde{P}_{K-1}$
E_{K-1}
Encoder Layer $\mathcal{V}_K$
Image Encoder $\mathcal{V}$

Figure 8: MaPLe Architecture

Learnable tokens in the initial layers of both vision and language branches are introduced and the multi-modal hierarchical prompts utilize the knowledge embedded in CLIP model to effectively learn task relevant contextual representations .

```
class TextEncoder(nn.Module):
    def __init__(self, clip_model):
        super().__init__()
        self.transformer = clip_model.transformer
        self.positional_embedding =
           clip_model.positional_embedding
        self.ln_final = clip_model.ln_final
        self.text_projection = clip_model.text_projection
        self.dtype = clip_model.dtype

    def forward(self, prompts, tokenized_prompts,
       compound_prompts_deeper_text):
        x = prompts + self.positional_embedding.type(self.dtype)
        x = x.permute(1, 0, 2)  # NLD -> LND
        # Pass as the list, as nn.sequential cannot process
           multiple arguments in the forward pass
        combined = [x, compound_prompts_deeper_text, 0]  # third
           argument is the counter which denotes depth of prompt
        outputs = self.transformer(combined)
        x = outputs[0]  # extract the x back from here
        x = x.permute(1, 0, 2)  # LND -> NLD
        x = self.ln_final(x).type(self.dtype)

        # x.shape = [batch_size, n_ctx, transformer.width]
        # take features from the eot embedding (eot_token is the
           highest number in each sequence)
        x = x[torch.arange(x.shape[0]),
           tokenized_prompts.argmax(dim=-1)] @
           self.text_projection

        return x


class MultiModalPromptLearner(nn.Module):
    def __init__(self, cfg, classnames, clip_model):
        super().__init__()
        n_cls = len(classnames)
        n_ctx = cfg.TRAINER.MAPLE.N_CTX
        ctx_init = cfg.TRAINER.MAPLE.CTX_INIT
        dtype = clip_model.dtype
        ctx_dim = clip_model.ln_final.weight.shape[0]
        clip_imsize = clip_model.visual.input_resolution
        cfg_imsize = cfg.INPUT.SIZE[0]
        # Default is 1, which is compound shallow prompting
        assert cfg.TRAINER.MAPLE.PROMPT_DEPTH >= 1, "For MaPLe,
           PROMPT_DEPTH should be >= 1"
        self.compound_prompts_depth =
           cfg.TRAINER.MAPLE.PROMPT_DEPTH  # max=12, but will
```

```
   create 11 such shared prompts
assert cfg_imsize == clip_imsize, f"cfg_imsize
   ({cfg_imsize}) must equal to clip_imsize
   ({clip_imsize})"

if ctx_init and (n_ctx) <= 4:
    # use given words to initialize context vectors
    ctx_init = ctx_init.replace("_", " ")
    n_ctx = n_ctx
    prompt = clip.tokenize(ctx_init)
    with torch.no_grad():
        embedding =
           clip_model.token_embedding(prompt).type(dtype)
    ctx_vectors = embedding[0, 1: 1 + n_ctx, :]
    prompt_prefix = ctx_init
else:
    # random initialization
    ctx_vectors = torch.empty(n_ctx, ctx_dim,
       dtype=dtype)
    nn.init.normal_(ctx_vectors, std=0.02)
    prompt_prefix = " ".join(["X"] * n_ctx)
print('MaPLe design: Multi-modal Prompt Learning')
print(f'Initial context: "{prompt_prefix}"')
print(f"Number of MaPLe context words (tokens): {n_ctx}")
# Linear layer so that the tokens will project to 512
   and will be initialized from 768
self.proj = nn.Linear(ctx_dim, 768)
self.proj.half()
self.ctx = nn.Parameter(ctx_vectors)
# Define the compound prompts for the deeper layers
self.compound_prompts_text =
   nn.ParameterList([nn.Parameter(torch.empty(n_ctx,
   512))
for _ in range(self.compound_prompts_depth - 1)])
    for single_para in self.compound_prompts_text:
        nn.init.normal_(single_para, std=0.02)
single_layer = nn.Linear(ctx_dim, 768)
self.compound_prompt_projections =
   _get_clones(single_layer, self.compound_prompts_depth
   - 1)

classnames = [name.replace("_", " ") for name in
   classnames]
name_lens = [len(_tokenizer.encode(name)) for name in
   classnames]
prompts = [prompt_prefix + " " + name + "." for name in
   classnames]

tokenized_prompts = torch.cat([clip.tokenize(p) for p in
   prompts])  # (n_cls, n_tkn)
```

```
        with torch.no_grad():
            embedding =
              clip_model.token_embedding(tokenized_prompts).type(dtype)

        self.register_buffer("token_prefix", embedding[:, :1,
           :])  # SOS
        self.register_buffer("token_suffix", embedding[:, 1 +
           n_ctx:, :])  # CLS, EOS

        self.n_cls = n_cls
        self.n_ctx = n_ctx
        self.tokenized_prompts = tokenized_prompts  #
           torch.Tensor
        self.name_lens = name_lens

    def construct_prompts(self, ctx, prefix, suffix, label=None):

        if label is not None:
            prefix = prefix[label]
            suffix = suffix[label]

        prompts = torch.cat(
            [
                prefix,  # (dim0, 1, dim)
                ctx,  # (dim0, n_ctx, dim)
                suffix,  # (dim0, *, dim)
            ],
            dim=1,
        )

        return prompts

    def forward(self):
        ctx = self.ctx

        if ctx.dim() == 2:
            ctx = ctx.unsqueeze(0).expand(self.n_cls, -1, -1)

        prefix = self.token_prefix
        suffix = self.token_suffix
        prompts = self.construct_prompts(ctx, prefix, suffix)

        visual_deep_prompts = []
        for index, layer in
           enumerate(self.compound_prompt_projections):
            visual_deep_prompts.append(layer(self.compound_prompts_text[i

        return prompts, self.proj(self.ctx),
           self.compound_prompts_text, visual_deep_prompts
```

```
class CustomCLIP(nn.Module):
    def __init__(self, cfg, classnames, clip_model):
        super().__init__()
        self.prompt_learner = MultiModalPromptLearner(cfg,
           classnames, clip_model)
        self.tokenized_prompts =
           self.prompt_learner.tokenized_prompts
        self.image_encoder = clip_model.visual
        self.text_encoder = TextEncoder(clip_model)
        self.logit_scale = clip_model.logit_scale
        self.dtype = clip_model.dtype

    def forward(self, image, label=None):
        tokenized_prompts = self.tokenized_prompts
        logit_scale = self.logit_scale.exp()

        prompts, shared_ctx, deep_compound_prompts_text,
           deep_compound_prompts_vision = self.prompt_learner()
        text_features = self.text_encoder(prompts,
           tokenized_prompts, deep_compound_prompts_text)
        image_features =
           self.image_encoder(image.type(self.dtype),
           shared_ctx, deep_compound_prompts_vision)

        image_features = image_features /
           image_features.norm(dim=-1, keepdim=True)
        text_features = text_features /
           text_features.norm(dim=-1, keepdim=True)
        logits = logit_scale * image_features @ text_features.t()

        if self.prompt_learner.training:
            return F.cross_entropy(logits, label)

        return logits


def _get_clones(module, N):
    return nn.ModuleList([copy.deepcopy(module) for i in
       range(N)])


@TRAINER_REGISTRY.register()
class MaPLe(TrainerX):
    def check_cfg(self, cfg):
        assert cfg.TRAINER.MAPLE.PREC in ["fp16", "fp32", "amp"]

    def build_model(self):
        cfg = self.cfg
        classnames = self.dm.dataset.classnames
```

```
print(f"Loading CLIP (backbone:
   {cfg.MODEL.BACKBONE.NAME})")
clip_model = load_clip_to_cpu(cfg)

if cfg.TRAINER.MAPLE.PREC == "fp32" or
   cfg.TRAINER.MAPLE.PREC == "amp":
    # CLIP's default precision is fp16
    clip_model.float()

print("Building custom CLIP")
self.model = CustomCLIP(cfg, classnames, clip_model)

print("Turning off gradients in both the image and the
   text encoder")
name_to_update = "prompt_learner"

for name, param in self.model.named_parameters():
    if name_to_update not in name:
        # Make sure that VPT prompts are updated
        if "VPT" in name:
            param.requires_grad_(True)
        else:
            param.requires_grad_(False)

# Double check
enabled = set()
for name, param in self.model.named_parameters():
    if param.requires_grad:
        enabled.add(name)
print(f"Parameters to be updated: {enabled}")

if cfg.MODEL.INIT_WEIGHTS:
    load_pretrained_weights(self.model,
       cfg.MODEL.INIT_WEIGHTS)

self.model.to(self.device)

self.optim = build_optimizer(self.model, cfg.OPTIM)
self.sched = build_lr_scheduler(self.optim, cfg.OPTIM)
self.register_model("MultiModalPromptLearner",
   self.model, self.optim, self.sched)

self.scaler = GradScaler() if cfg.TRAINER.MAPLE.PREC ==
   "amp" else None


device_count = torch.cuda.device_count()
if device_count > 1:
    print(f"Multiple GPUs detected
                (n_gpus={device_count}), use all of them!")
            self.model = nn.DataParallel(self.model)

    def forward_backward(self, batch):
        image, label = self.parse_batch_train(batch)

        model = self.model
        optim = self.optim
        scaler = self.scaler

        prec = self.cfg.TRAINER.MAPLE.PREC
        if prec == "amp":
            with autocast():
                loss = model(image, label)
            optim.zero_grad()
            scaler.scale(loss).backward()
            scaler.step(optim)
            scaler.update()
        else:
            loss = model(image, label)
            optim.zero_grad()
            loss.backward()
            optim.step()

        loss_summary = {"loss": loss.item()}

        if (self.batch_idx + 1) == self.num_batches:
            self.update_lr()

        return loss_summary

    def parse_batch_train(self, batch):
        input = batch["img"]
        label = batch["label"]
        input = input.to(self.device)
        label = label.to(self.device)
        return input, label

    def load_model(self, directory, epoch=None):
        if not directory:
            print("Note that load_model() is skipped as no
                pretrained model is given")
            return

        names = self.get_model_names()

        # By default, the best model is loaded
        model_file = "model-best.pth.tar"

        if epoch is not None:
```

```
        model_file = "model.pth.tar-" + str(epoch)

    for name in names:
        model_path = osp.join(directory, name, model_file)

        if not osp.exists(model_path):
            raise FileNotFoundError('Model not found at
               "{}"'.format(model_path))

        checkpoint = load_checkpoint(model_path)
        state_dict = checkpoint["state_dict"]
        epoch = checkpoint["epoch"]

        # Ignore fixed token vectors
        if "prompt_learner.token_prefix" in state_dict:
            del state_dict["prompt_learner.token_prefix"]

        if "prompt_learner.token_suffix" in state_dict:
            del state_dict["prompt_learner.token_suffix"]

        print("Loading weights to {} " 'from "{}" (epoch =
           {})'.format(name, model_path, epoch))

        self._models[name].load_state_dict(state_dict,
           strict=False)
```

8 Leveraging Prompt Learning Architecture for CD-FSL

This study of the architecture and corresponding implementation of prompt engineering architecture brings an important cross modal and contextual perspective to the CD-FSL task. In the CD-FSL environment where there is significant domain gap integrating and adapting prompt learning setting helps to bridge this gap and improves generalization ability and accuracy in navigating to unseen novel classes.

The text prompts provide the necessary contextual embeddings that provide better insights for the classification. This paradigm also proves to be efficient in learning transferable representations that show remarkable adaptability in novel classes.

9 Future Work

Further enhancements in finetuning the pretrained model such as Self Regulating Prompts (PromptSRC) with explicit self-regulating constraints improves on base classes as well as shows improvements on novel classes. Prompts are also used in approaches such as PromptStyler which simulates various distribution shifts in the joint space by synthesizing diverse styles without using any images to deal with source-free domain generalization. Methods such as Prompting to Disentangle Domain Knowledge (PRoD) that generates a learnable domain general prompt and a domain specific prompt generated for the domain of interest can be explored and adapted in the CD-FSL setting.

10 References

A Survey of Deep Visual Cross-Domain Few-Shot Learning - Wenjian Wang, Lijuan Duan, Yuxi Wang, Junsong Fan, Zhi Gong and Zhaoxiang Zhang

StyleAdv: Meta Style Adversarial Training for Cross-Domain Few-Shot Learning - Yuqian Fu, Yu Xie, Yanwei Fu, Yu-Gang Jiang

Cross-domain Few-shot Learning with Task-specific Adapters - Wei-Hong Li, Xialei Liu, and Hakan Bilen

An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale - Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, Neil Houlsby

A Transductive Multi-Head Model for Cross-Domain Few-Shot Learning - Jianan Jiang, Zhenpeng Li, Yuhong Guo, Jieping Ye

Learning Transferable Visual Models From Natural Language Supervision - Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, Ilya Sutskever

Learning to Prompt for Vision-Language Models - Kaiyang Zhou, Jingkang Yang , Chen Change Loy, Ziwei Liu

Conditional Prompt Learning for Vision-Language Models- Kaiyang Zhou, Jingkang Yang, Chen Change Loy, Ziwei Liu

MaPLe: Multi-modal Prompt Learning - Muhammad Uzair Khattak, Hanoona Rasheed, Muhammad Maaz, Salman Khan, Fahad Shahbaz Khan

ProD: Prompting-To-Disentangle Domain Knowledge for Cross-Domain Few-Shot Image Classification - Tianyi Ma, Yifan Sun, Zongxin Yang, Yi Yang

Multimodality Helps Unimodality: Cross-Modal Few-Shot Learning with Multimodal Models - Zhiqiu Lin, Samuel Yu, Zhiyi Kuang, Deepak Pathak, Deva Ramanan

Self-regulating Prompts: Foundational Model Adaptation without Forgetting - Muhammad Uzair Khattak, Syed Talal Wasim, Muzammal Naseer, Salman Khan, Ming-Hsuan Yang, Fahad Shahbaz Khan

PromptStyle: Controllable Style Transfer for Text-to-Speech with Natural Language Descriptions - Guanghou Liu, Yongmao Zhang, Yi Lei, Yunlin Chen, Rui Wang, Zhifei Li, Lei Xie