Understanding CLIP by OpenAI

by ANKIT SACHAN

CLIP By OPEN-AI

Introduction

Nearly all state-of-the-art visual perception algorithms rely on the same formula:

(1) pretrain a convolutional network on a large, manually annotated image classification dataset

(2) finetune the network on a smaller, task-specific dataset.

This technique has been widely used for several years and has led to impressive improvements on numerous tasks.

State-of-the-art visual perception models for a wide range of tasks rely on supervised pretraining. ImageNet classification is the de facto pretraining task for these models. Yet, ImageNet is now nearly ten years old and is by modern standards “small”. Even so, relatively little is known about the behavior of pretraining with datasets that are multiple orders of magnitude larger.

Even after all this, standard computer vision models have trouble generalizing to unseen test cases. This raises questions about the entire deep learning approach towards computer vision.

Background

CLIP (Contrastive Language–Image Pre-training) deviates from the standard practice of fine-tuning a pretrained model by taking the path of zero-shot learning. As described in the previous blog on DALL-E, zero-shot learning is the ability of the model to perform tasks that it was not explicitly programmed to do.

In 2016, Li et al. [1] demonstrated that using natural language-based predictions, their model achieved about 11.4% zero-shot accuracy on the imagenet dataset. They fine-tuned a 34 layer deep residual network that was pretrained on the imagenet dataset. Thirty million English comments from Flickr were used as a dataset to perform supervised learning. Li et al. trained their model to output n-grams for a given image.

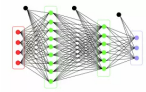
However, 11.4% accuracy is far from the current state of the art, i.e., 84% accuracy (Xie et al., 2020). It is even below the 50% accuracy of classic computer vision approaches (Deng et al., 2012). This shows us that using **just** raw text as weakly supervised learning methods does not yield good results.

On the other hand, Mahajan et al. (2018) showed that predicting ImageNet-related hashtags on Instagram images is an effective pre-training task. When fine-tuned to ImageNet, these pre-trained models increased accuracy by over 5% and improved the overall state of the art at the time. It is evident that there is a thin line between using finely annotated images to train your network and using practically unlimited raw text to train your network.

MOST POPULAR

Tensorflow Tutorial 2: image classifier using convolutional neural network


A quick complete tutorial to save and restore Tensorflow models


ResNet, AlexNet, VGGNet, Inception: Understanding various architectures of Convolutional Networks


Zero to Hero: Guide to Object Detection using Deep Learning: ...

Keras tutorial: Practical guide from getting started to developing complex ...

Authors of CLIP created a new dataset consisting of 400 million training examples (images, text) and trained a simplified version of the ConVIRT model, i.e., the CLIP model, on their novel dataset. This model was trained from scratch and had similarities with the GPT model. It had knowledge about geo-localization, OCR, action recognition, and much more.

CLIP's core idea

The core idea of the CLIP paper is essentially to learn visual representation from the massive corpus of natural language data. The paper showed that a simple pre-training task is sufficient to achieve a competitive performance boost in zero-shot learning.

The objective of the CLIP model can be understood as following:

Given an image, a set of 32,768 randomly sampled text snippets was paired with it in our dataset. For example, given a task to predict a number from an image, the model is likely to predict that "the number is one" or, "the number is two", or "the number is xyz" and so on.

The model would have to learn the extensive connections between visual data and their related words from the language data to achieve this. This is the intuition behind using a massive corpus of natural language data and their paired images to train the model.

Training objective

State-of-the-art computer vision systems use enormous amounts of computational resources. Mahajan et al. (2018) required 19 GPU years to train their ResNeXt101-32x48d and Xie et al. (2020) required 33 TPUv3 core-years to train their Noisy Student EfficientNet-L2.

Initially, the authors jointly trained an image CNN and text transformer from scratch to predict the caption of an image. However, this approach turned out to be highly compute intensive. A 63 million parameter transformer language model, which already uses twice the compute of its ResNet-50 image encoder, learns to recognize ImageNet classes three times slower than a much simpler baseline that predicts a bag-ofwords encoding of the same text.

On further introspection, this approach was found to be flawed because of the predictions that were expected from the transformer. Here, the transformer was required to output the hashtags/comments as it is rather than letting the CNN focus on the important visual data.

To overcome this, a contrastive objective was adopted which increased the efficiency of the CLIP model by 4x times. In other words say that we are given N (image, text) pairs of training examples. The CLIP model consists of a text and an image encoder which encodes textual and visual information into a multimodal embedding space. Now, the aim of the model is to increase the cosine similarity score of images and text which is actually associated which in this case there are N such pairs. On the other hand, the model also tries to minimize the similarity between images and texts which do not occur together which in this case would be N^2 – N such pairs.

This would make more sense once you go through the attached python code snippet.

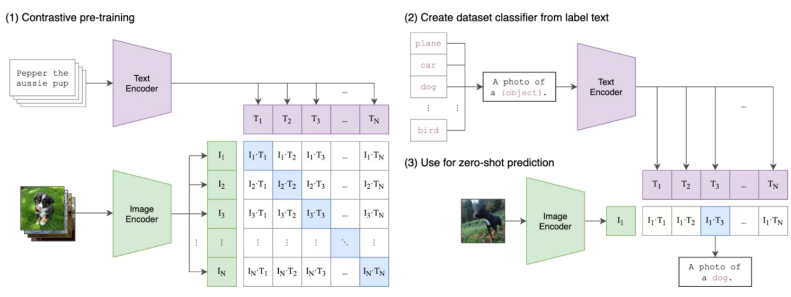
```
# image_encoder - ResNet or Vision Transformer PyTorch Module
# text_encoder - CBOW or Text Transformer PyTorch Module
# I[n, h, w, c] - minibatch of aligned images
# T[n, l] - minibatch of aligned texts
# W_i[d_i, d_e] - learned proj of image to embed
# W_t[d_t, d_e] - learned proj of text to embed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



As you can see here, the contrastive pretraining involves maximising cosine similarity of encodings on the diagonal of the N*N matrix since they are the actual image, text pairs.

In the second figure, the CLIP model can be seen in action by correctly predicting the dog by maximising the similarity between the word dog and the visual information.

Model Architecture

Here, the authors have used two different backbones (Resnet50 and Vision Transformer (ViT)) for the image encoder and a Transformer as the backbone for the text encoder.

The largest ResNet model, RN50x64, took 18 days to train on 592 V100 GPUs while the largest Vision Transformer took 12 days on 256 V100 GPUs.

Let us understand the code for the CLIP model function by function to gain a better insight into the model architecture.

The model is instanstiated and all necessary attributes are assigned by the constructor call. By specifying the vision_layers attribute as a tuple, list type of an object, we can use the resnet architecture as the visual representation encoder's backbone. In any other case, the model instantiates the Visual Transformer as the backbone. Embed_dim is used to define the dimensions of the embedding space. Width and layer parameters are used to specify the width and number of layers of the respective bacbone networks.

```

class CLIP(nn.Module):
    def __init__(self,
                 embed_dim: int,
                 # vision
                 image_resolution: int,
                 vision_layers: Union[Tuple[int, int, int, int], int],
                 vision_width: int,
                 vision_patch_size: int,
                 # text
                 context_length: int,
                 vocab_size: int,
                 transformer_width: int,
                 transformer_heads: int,
                 transformer_layers: int
                 ):
        super().__init__()

        self.context_length = context_length

        if isinstance(vision_layers, (tuple, list)):
            vision_heads = vision_width * 32 // 64
            self.visual = ModifiedResNet(
                layers=vision_layers,
                output_dim=embed_dim,
                heads=vision_heads,
                input_resolution=image_resolution,
                width=vision_width
            )

        else:
            vision_heads = vision_width // 64
            self.visual = VisualTransformer(
                input_resolution=image_resolution,
                patch_size=vision_patch_size,
                width=vision_width,
                layers=vision_layers,
                heads=vision_heads,
                output_dim=embed_dim
            )

        self.transformer = Transformer(
            width=transformer_width,
            layers=transformer_layers,
            heads=transformer_heads,
            attn_mask=self.build_attention_mask()
        )

        self.vocab_size = vocab_size
        self.token_embedding = nn.Embedding(vocab_size, transformer_width)
        self.positional_embedding = nn.Parameter(torch.empty(self.context_length, transformer_width
        self.ln_final = LayerNorm(transformer_width)
        self.text_projection = nn.Parameter(torch.empty(transformer_width, embed_dim))
        self.logit_scale = nn.Parameter(torch.ones([]) * np.log(1 / 0.07))

        self.initialize_parameters()
```

In this function, we are simply initializing parameters of the backbone networks. Note that we are not yet assigning pretrained weights to the backbone nets.

```

def initialize_parameters(self):
        nn.init.normal_(self.token_embedding.weight, std=0.02)
        nn.init.normal_(self.positional_embedding, std=0.01)

        if isinstance(self.visual, ModifiedResNet):
            if self.visual.attnpool is not None:
                std = self.visual.attnpool.c_proj.in_features ** -0.5
                nn.init.normal_(self.visual.attnpool.q_proj.weight, std=std)
                nn.init.normal_(self.visual.attnpool.k_proj.weight, std=std)
                nn.init.normal_(self.visual.attnpool.v_proj.weight, std=std)
                nn.init.normal_(self.visual.attnpool.c_proj.weight, std=std)

            for resnet_block in [self.visual.layer1, self.visual.layer2, self.visual.layer3, self.v
                for name, param in resnet_block.named_parameters():
                    if name.endswith("bn3.weight"):
                        nn.init.zeros_(param)

        proj_std = (self.transformer.width ** -0.5) * ((2 * self.transformer.layers) ** -0.5)
        attn_std = self.transformer.width ** -0.5
        fc_std = (2 * self.transformer.width) ** -0.5
```

```
        for block in self.transformer.resblocks:
            nn.init.normal_(block.attn.in_proj_weight, std=attn_std)
            nn.init.normal_(block.attn.out_proj.weight, std=proj_std)
            nn.init.normal_(block.mlp.c_fc.weight, std=fc_std)
            nn.init.normal_(block.mlp.c_proj.weight, std=proj_std)

        if self.text_projection is not None:
            nn.init.normal_(self.text_projection, std=self.transformer.width ** -0.5)

    def build_attention_mask(self):
        # lazily create causal attention mask, with full attention between the vision tokens
        # pytorch uses additive attention mask; fill with -inf
        mask = torch.empty(self.context_length, self.context_length)
        mask.fill_(float("-inf"))
        mask.triu_(1)  # zero out the lower diagonal
        return mask
```

Running a forward pass on the image encoder

```

    def encode_image(self, image):
        return self.visual(image.type(self.dtype))
```

Running a forward pass on the text encoder

```

    def encode_text(self, text):
        x = self.token_embedding(text).type(self.dtype)  # [batch_size, n_ctx, d_model]

        x = x + self.positional_embedding.type(self.dtype)
        x = x.permute(1, 0, 2)  # NLD -> LND
        x = self.transformer(x)
        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]), text.argmax(dim=-1)] @ self.text_projection

        return x
```

The forward pass of the clip model involves running a forward pass through the text and image encoder network. These embedded features are then normalised and used as input to the cosine similarity. Finally the cosine similarity is computed and returned as logits.

```
def forward(self, image, text):
        image_features = self.encode_image(image)
        text_features = self.encode_text(text)

        # normalized features
        image_features = image_features / image_features.norm(dim=-1, keepdim=True)
        text_features = text_features / text_features.norm(dim=-1, keepdim=True)

        # cosine similarity as logits
        logit_scale = self.logit_scale.exp()
        logits_per_image = logit_scale * image_features @ text_features.t()
        logits_per_text = logit_scale * text_features @ image_features.t()

        # shape = [global_batch_size, global_batch_size]
        return logits_per_image, logits_per_text
```

Conclusion

CLIP is highly effective in learning visual representations through the freely available massive corpus of text data. It is known that by training giant neural networks on such a huge amount of data, zero-shot learning tends to take place. In fact, the model was also able to recognize a few classes that were not even a part of the training set. By making use of the contrastive objective function and visual transformer, OPEN-AI has developed a highly resilient and compute efficient model.

Furthermore, on carrying out quantitative experiments, the authors found that CLIP model is significantly more flexible than the current SOTA by validating the scores on 30 different datasets. These tasks included OCR, geolocalisation and action recognition. The best CLIP model outperformed the best imagenet model on 20 out of the 26 datasets that were tested by the team.

CLIP also has its limitations on the other hand. It struggles with slightly complex tasks such as counting the number of objects in an image, predicting how far an object is from the camera (no sense of depth perception) and telling differences between similar objects. Even though it has a great zero shot accuracy on OCR, it performs poorly on classifying the MNIST dataset at an accuracy of 88%.

Finally, we can conclude by saying that CLIP is a ground breaking work in terms of reducing efforts to find well annotated images dataset to perform image classification. Since it does not require task specific training data, we can keep feeding it massive amounts of raw text data and it would slowly get better and better at more unrelated tasks.

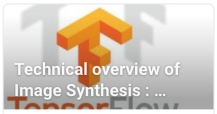
ALSO ON **CV-TRICKS.COM**

3 years ago

In this quick Tensorflow tutorial, you shall learn what's a Tensorflow 2.0 ...

3 years ago

In this quick Tensorflow tutorial, you shall learn what's a Tensorflow ...

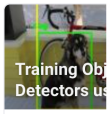


6 months ago

Looking at the inner workings of Stable Diffusion architecture for Image ...

3 years ago

Alexnet, which started the deep learning revolution, was loosely based on a ...



2 years ago

In this blog pc describe how Tensorflow Ot