LoRA Fine-tuning on CLIP from scratch

Speeding up fine tuning with reduced memory consumption using low-rank matrices

MaxIconway · Follow

Published in Toward Humanoids · 13 min read · Oct 16, 2024

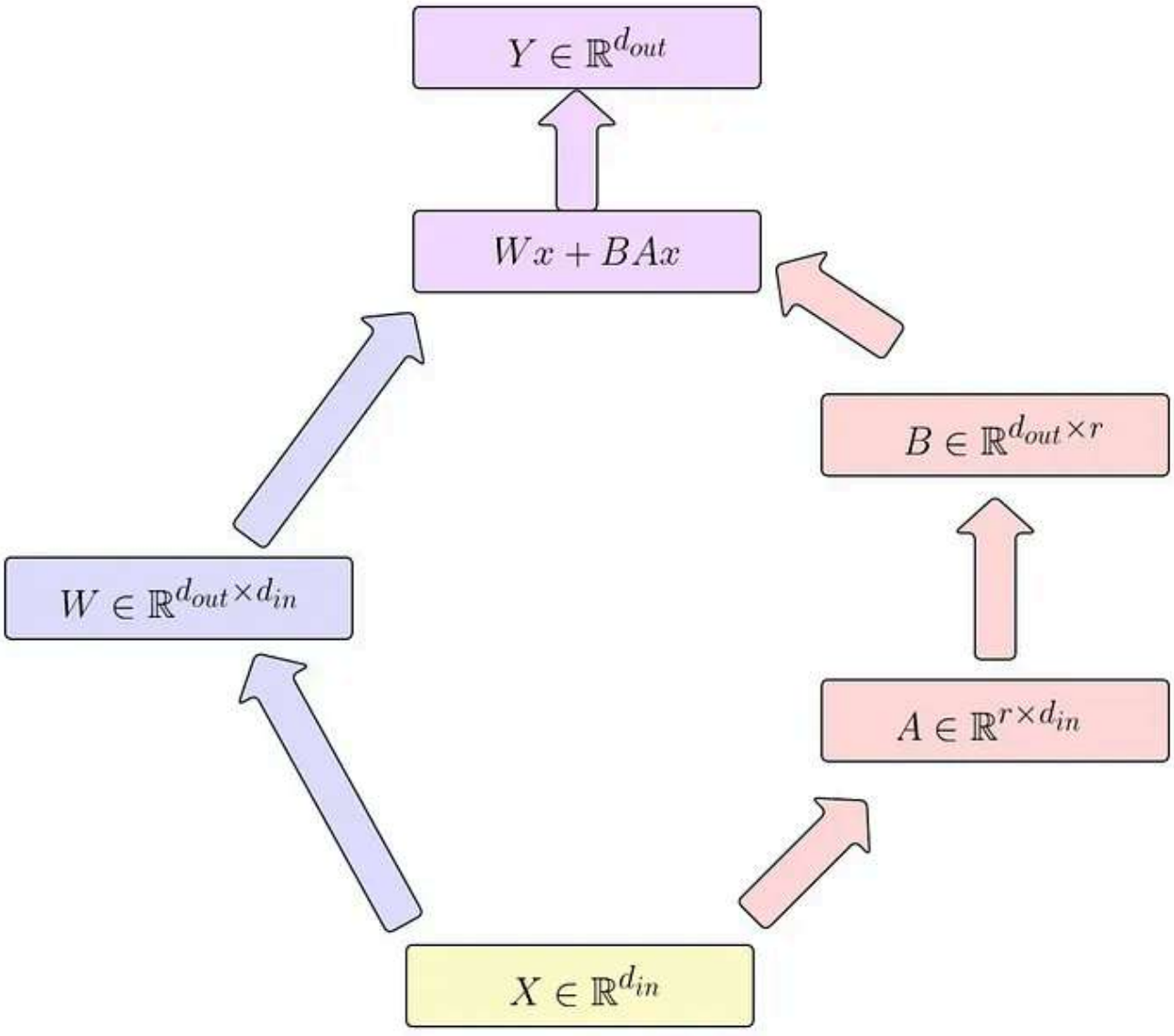
In this article we will show how to modify a basic CLIP implementation to support LoRA fine tuning. We first train a CLIP model from scratch on the Caltech101 dataset and then fine tuned on the Fashion MNIST dataset. We achieve equivalent accuracy to a model trained from scratch on Fashion MNIST while training significantly less parameters on Fashion MNIST, demonstrating how fine-tuning on a reduced rank matrix can reduce memory consumption and lead to faster fine-tuning.

This work builds on Matt Nguyen's article Building CLIP from Scratch, please familiarize yourself with that article before reading this one.

Building CLIP From Scratch

Open World Object Recognition on the FashionMNIST Dataset

medium.com



The original LoRA: Low-Rank Adaptation of Large language Models paper is available here:

LoRA: Low-Rank Adaptation of Large Language Models

An important paradigm of natural language processing consists of large-scale pre-training on general domain data and...

arxiv.org

Low Rank Adaption or LoRA is a fine tuning technique that works on a simple premise. For a pretrained matrix W

$$W \in \mathbb{R}^{M \times N}$$

Who's forward pass is computed as

$$Wx$$

traditional fine tuning would modify W as a function of its gradient with respect to a loss function

$$W \leftarrow W - \eta \nabla_W L(Wx)$$

however in LoRA fine tuning we define two additional matrices that when multiplied have the same dimension as W.

$$A \in \mathbb{R}^{R \times N}$$

$$B \in \mathbb{R}^{M \times R}$$

$$BA \in \mathbb{R}^{M \times N}$$

Note that BA is an MxN matrix with rank R. During fine tuning then we do not update the pretrained matrix W, instead we learn matrices A and B that minimize the loss function when A and B are integrated into the forward pass on model as

$$Wx + BAx$$

We then learn A and B via gradient decent

$$B \leftarrow B - \eta \nabla_B L(Wx + BAx)$$

$$A \leftarrow A - \eta \nabla_A L(Wx + BAx)$$

While we could add these LoRA update to every component of Matt Nguyens article, in the spirit of the original LoRA paper we will only fine tune the query matrix Q in the attention mechanism.

To enable this functionality we must modify the AttentionHead class from Matt's original article, we modify the initialization of the attention head to include a lora_rank parameter and instantiate all the different A and B matrices we need.

```
def __init__(self, width, head_size, lora_rank):
    super().__init__()
    self.head_size = head_size

    self.query = nn.Linear(width, head_size)
    self.key = nn.Linear(width, head_size)
    self.value = nn.Linear(width, head_size)

    self.qA = nn.Linear(width, lora_rank)
```

```
        self.qB = nn.Linear(lora_rank, head_size)

        #self.kA = nn.Linear(width, lora_rank)
        #self.kB = nn.Linear(lora_rank, head_size)

        #self.vA = nn.Linear(width, lora_rank)
        #self.vB = nn.Linear(lora_rank, head_size)

        #nn.init.constant_(self.qB.weight, 0)
        #nn.init.constant_(self.qB.bias, 0)

        #nn.init.constant_(self.kB.weight, 0)
        #nn.init.constant_(self.kB.bias, 0)

        #nn.init.constant_(self.vB.weight, 0)
        #nn.init.constant_(self.vB.bias, 0)

        self.set_lora_mode(False)
```

Notice we set the weights and bias in the B matrix to 0 so that initially BAx contributes nothing to the forward pass. Next we define a new method for the AttentionHead class called set_lora_mode.

```
    def set_lora_mode(self, mode):

        self.query.requires_grad_(not mode)
        self.key.requires_grad_(not mode)
        self.value.requires_grad_(not mode)

        self.qA.requires_grad_(mode)
        self.qB.requires_grad_(mode)

        #self.kA.requires_grad_(mode)
        #self.kB.requires_grad_(mode)

        #self.vA.requires_grad_(mode)
        #self.vB.requires_grad_(mode)
```

When set LoRA mode is true the query, key and value matrices are frozen while the A and B matrices can be learned, and when LoRA mode is false the A and B matrices are frozen and the query, key, value matrices can be learned.

Finally we must update the forward pass to use the A and B matrices, these will be learned via back propagation in pytorch.

```
def forward(self, x, mask=None):
    # Obtaining Queries, Keys, and Values
    Q = self.query(x) + self.qB(self.qA(x))
    K = self.key(x)# + self.kB(self.kA(x))
    V = self.value(x)# + self.vB(self.vA(x))

    # Dot Product of Queries and Keys
    attention = Q @ K.transpose(-2,-1)
    #print(f"attention matrix: {attention.shape}")

    # Scaling
    attention = attention / (self.head_size ** 0.5)

    # Applying Attention Mask
    if mask is not None:
        attention = attention.masked_fill(mask == 0, float("-inf"))

    attention = torch.softmax(attention, dim=-1)

    attention = attention @ V
    #print(f"attention vectors: {attention.shape}")

    return attention
```

Note that if we wanted to fine tune the V and K matrices we can just uncomment the corresponding code

Next we need to update the MultiHeadAttention class. Again we add a new argument lora_rank to the initialization and pass it on to each attention head to instantiate our A and B matrices

```
def __init__(self, width, n_heads, lora_rank):
    super().__init__()
    self.head_size = width // n_heads

    self.W_o = nn.Linear(width, width)

    #self.oA = nn.Linear(width, lora_rank)
    #self.oB = nn.Linear(lora_rank, width)

    #nn.init.constant_(self.oB.weight, 0)
    #nn.init.constant_(self.oB.bias, 0)

    self.heads = nn.ModuleList([AttentionHead(width, self.head_size, lora_rank)
    self.set_lora_mode(False)
```

We also add a set_lora_mode method to the MultiHeadAttention Class

```
def set_lora_mode(self, mode):
    for head in self.heads:
      head.set_lora_mode(mode)

    self.W_o.requires_grad_(not mode)
    #self.oA.requires_grad_(mode)
    #self.oB.requires_grad_(mode)
```

and just like in the attention head we modify the forward pass

```
    def forward(self, x, mask=None):
        # Combine attention heads
        out = torch.cat([head(x, mask=mask) for head in self.heads], dim=-1)

        out = self.W_o(out)# + self.oB(self.oA(out))

        return out
```

Again we could fine tune the out matrix by uncommenting the corresponding code.

With the AttentionHead and MultiHeadAttention classes modified we just need to fix some plumbing so that the classes using these classes can use our new functionality.

The transformer encoder class gets a new lora_rank initialization argument and a set_lora_mode method.

```
class TransformerEncoder(nn.Module):
    def __init__(self, width, n_heads, lora_rank, r_mlp=4):
        super().__init__()
        self.width = width
        self.n_heads = n_heads

        # Sub-Layer 1 Normalization
        self.ln1 = nn.LayerNorm(width)

        # Multi-Head Attention
        self.mha = MultiHeadAttention(width, n_heads, lora_rank)

        # Sub-Layer 2 Normalization
        self.ln2 = nn.LayerNorm(width)

        # Multilayer Perception
        self.mlp = nn.Sequential(
            nn.Linear(self.width, self.width*r_mlp),
            nn.GELU(),
```

```
            nn.Linear(self.width*r_mlp, self.width)
        )
        self.set_lora_mode(False)
    def set_lora_mode(self, mode):
      self.mha.set_lora_mode(mode)
      self.mlp.requires_grad_(not mode)

    def forward(self, x, mask=None):
        # Residual Connection After Sub-Layer 1
        x = x + self.mha(self.ln1(x), mask=mask)

        # Residual Connection After Sub-Layer 2
        x = x + self.mlp(self.ln2(x))

        return x
```

The TextEncoder Class also gets a lora_rank initialization argument and a set_lora_mode method

```
class TextEncoder(nn.Module):
    def __init__(self, vocab_size, width, max_seq_length, n_heads, n_layers, emb
        super().__init__()

        self.max_seq_length = max_seq_length  # Maximum length of input sequence

        self.encoder_embedding = nn.Embedding(vocab_size, width) # Embedding Tab

        self.positional_embedding = PositionalEmbedding(width, max_seq_length)

        self.encoder = nn.ModuleList([TransformerEncoder(width, n_heads, lora_ra

        # learned proj of image to embed
        self.projection = nn.Parameter(torch.randn(width, emb_dim))
        self.set_lora_mode(False)

    def set_lora_mode(self, mode):
      self.encoder_embedding.requires_grad_(not mode)
      self.positional_embedding.requires_grad_(not mode)
      for encoder in self.encoder:
        encoder.set_lora_mode(mode)
      self.projection.requires_grad_(not mode)

    def forward(self, text, mask):
```

```
        # Text Embedding
        x = self.encoder_embedding(text)

        # Positional Embedding
        x = self.positional_embedding(x)

        # Transformer Encoder
        for encoder_layer in self.encoder:
            x = encoder_layer(x, mask=mask)
        #print(f"x:{x.size()}")
        #print(f"text: {text.size()}")
        #print(f"mask:{mask.size()}")
        # Takes features from the EOT Embedding
        x = x[torch.arange(text.shape[0]),torch.sub(torch.sum(mask[:,0],dim=1),1

        # joint multimodal embedding
        if self.projection is not None:
            x = x @ self.projection

        x = x / torch.norm(x, dim=-1, keepdim=True)

        return x
```

Just like the text encoder and transformer encoder we give the image encoder a lora_rank argument and set_lora_mode method

```
class ImageEncoder(nn.Module):
    def __init__(self, width, img_size, patch_size, n_channels, n_layers, n_head
        super().__init__()

        assert img_size[0] % patch_size[0] == 0 and img_size[1] % patch_size[1]
        assert width % n_heads == 0, "width must be divisible by n_heads"

        self.n_patches = (img_size[0] * img_size[1]) // (patch_size[0] * patch_s

        self.max_seq_length = self.n_patches + 1

        # Patch Embedding
        self.linear_project = nn.Conv2d(n_channels, width, kernel_size=patch_siz

        # Classification Token
        self.cls_token = nn.Parameter(torch.randn(1, 1, width))
```

```
        self.positional_embedding = PositionalEmbedding(width,self.max_seq_lengt

        self.encoder = nn.ModuleList([TransformerEncoder(width,n_heads, lora_ran

        # learned proj of image to embed
        self.projection = nn.Parameter(torch.randn(width, emb_dim))
        self.set_lora_mode(False)


    def set_lora_mode(self, mode):
      self.linear_project.requires_grad_(not mode)
      self.cls_token.requires_grad_(not mode)
      for encoder in self.encoder:
        encoder.set_lora_mode(mode)
      self.projection.requires_grad_(not mode)

    def forward(self,x):
        # Patch Embedding
        x = self.linear_project(x)
        x = x.flatten(2).transpose(1, 2)

        # Positional Embedding
        x = torch.cat((self.cls_token.expand(x.size()[0], -1, -1),x), dim=1)
        x = self.positional_embedding(x)

        # Transformer Encoder
        for encoder_layer in self.encoder:
            x = encoder_layer(x)

        # Takes Class Tokens
        x = x[:, 0, :]

        # joint multimodal embedding
        if self.projection is not None:
            x = x @ self.projection

        x = x / torch.norm(x, dim=-1, keepdim=True)

        return x
```

Finally we update the CLIP class to have a lora rank and set_lora_mode method

```
class CLIP(nn.Module):
    def __init__(self, emb_dim, vit_width, img_size, patch_size, n_channels, vit
        super().__init__()

        self.image_encoder = ImageEncoder(vit_width, img_size, patch_size, n_cha

        self.text_encoder = TextEncoder(vocab_size, text_width, max_seq_length,

        self.temperature = nn.Parameter(torch.ones([]) * np.log(1 / 0.07))

        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu
        print(f"useing: {self.device}")
        self.set_lora_mode(False)

    def set_lora_mode(self, mode):
      self.image_encoder.set_lora_mode(mode)
      self.text_encoder.set_lora_mode(mode)


    def forward(self,image,text, mask=None):
        I_e = self.image_encoder(image)
        T_e = self.text_encoder(text, mask=mask)

        # scaled pairwise cosine similarities [n, n]
        logits = (I_e @ T_e.transpose(-2,-1)) * torch.exp(self.temperature)

        # symmetric loss function
        labels = torch.arange(logits.shape[0]).to(self.device)

        loss_i = nn.functional.cross_entropy(logits.transpose(-2,-1), labels)
        loss_t = nn.functional.cross_entropy(logits, labels)

        loss = (loss_i + loss_t) / 2

        return loss
```

With all that implemented we are ready to run our experiments, the only changes we make to the FashionMNIST dataset is that we modify the transform to include resizing and to create a validation split.

```
img_size = (128,128)
max_text_seq_length = 128

dataset = load_dataset("fashion_mnist")
dataset.set_format(type='torch')
num_train_samples = len(dataset['train'])
split_10 = int(0.1 * num_train_samples)
split_90 = num_train_samples - split_10
mnist_train_dataset, mnist_val_dataset = random_split(dataset['train'], [split_9
mnist_test_dataset = dataset["test"]


class FashionMNIST(Dataset):
    def __init__(self, split="train"):
        if split=="train":
            self.dataset = mnist_train_dataset
            self.split = "train"
            self.transform = T.Compose([
                T.Resize(img_size),
                #T.ToTensor(),
                T.ConvertImageDtype(torch.float32)  # Convert the image to float
            ])
        elif split=="val":
            self.dataset = mnist_val_dataset
            self.split = "val"
            self.transform = T.Compose([
                T.Resize(img_size),
                #T.ToTensor(),
                T.ConvertImageDtype(torch.float32)  # Convert the image to float
            ])
        else:
            self.dataset = mnist_test_dataset
            self.split = "test"
            self.transform = T.Compose([
                T.Resize(img_size),
                #T.ToTensor(),
                T.ConvertImageDtype(torch.float32)  # Convert the image to float
            ])

        #print(type(self.dataset))
        #print(dir(self.dataset))



        self.captions = {
            0: "t-shirt/top",
            1: "trousers",
            2: "pullover",
```

```
            3: "dress",
            4: "coat",
            5: "sandal",
            6: "shirt",
            7: "sneaker",
            8: "bag",
            9: "ankle boot"
        }
        self.captions = {k: f"an image of {v}" for k, v in self.captions.items()


    def __len__(self):
        return len(self.dataset)

    def __getitem__(self,i):
        img = self.dataset[i]["image"]
        img = self.transform(img)

        cap, mask = tokenizer(self.captions[self.dataset[i]["label"].item()], ma

        mask = mask.repeat(len(mask),1)

        return {"image": img, "caption": cap, "mask": mask}
```

Next we create a wrapper for the Caltech101 dataset that outputs data in the same format as the MNIST Fashion dataset used in Matt's Article

```
transform = T.Compose([
              T.Resize(img_size),  # Resize to a fixed size (e.g., for models li
              T.Grayscale(num_output_channels=1),  # Convert to grayscale with 1
              T.ToTensor(),
            ])

dataset = Caltech101(root="./", target_type='category', transform=transform, dow
train_size = int(0.7 * len(dataset))
val_size = int(0.2 * len(dataset))
test_size = len(dataset) - train_size - val_size
cal_train_dataset, cal_val_dataset, cal_test_dataset = random_split(dataset, [tr

class Caltech101_Wrapper(Dataset):
    def __init__(self, split="train"):
```

```
        # Define transformations (you can adjust these based on model requiremen

        if split == "train":
          self.dataset = cal_train_dataset
        elif split == "val":
          self.dataset = cal_val_dataset
        else:
          self.dataset = cal_test_dataset



        # Load the Caltech101 dataset

        # Captions for the different object categories
        self.captions = {
          0: "accordion",
          1: "airplane",
          2: "anchor",
          3: "ant",
          4: "from Google backgrounds",
          5: "barrel",
          6: "bass fish",
          7: "beaver",
          8: "binoculars",
          9: "bonsai tree",
          10: "brain",
          11: "brontosaurus",
          12: "Buddha statue",
          13: "butterfly",
          14: "camera",
          15: "cannon",
          16: "car viewed from the side",
          17: "ceiling fan",
          18: "cellphone",
          19: "chair",
          20: "chandelier",
          21: "cougar's body",
          22: "cougar's face",
          23: "crab",
          24: "crayfish",
          25: "crocodile",
          26: "crocodile's head",
          27: "cup",
          28: "dalmatian dog",
          29: "dollar bill",
          30: "dolphin",
          31: "dragonfly",
          32: "electric guitar",
          33: "elephant",
          34: "emu",
```

```
    35: "euphonium instrument",
    36: "ewer",
    37: "face",
    38: "face in an easy-to-recognize pose",
    39: "ferry",
    40: "flamingo",
    41: "flamingo's head",
    42: "character Garfield",
    43: "gerenuk",
    44: "gramophone",
    45: "grand piano",
    46: "hawksbill turtle",
    47: "headphones",
    48: "hedgehog",
    49: "helicopter",
    50: "ibis bird",
    51: "inline skate",
    52: "Joshua tree",
    53: "kangaroo",
    54: "ketch sailboat",
    55: "lamp",
    56: "laptop",
    57: "leopard",
    58: "llama",
    59: "lobster",
    60: "lotus flower",
    61: "mandolin",
    62: "mayfly",
    63: "menorah",
    64: "metronome",
    65: "minaret",
    66: "motorbike",
    67: "nautilus shell",
    68: "octopus",
    69: "okapi",
    70: "pagoda",
    71: "panda bear",
    72: "pigeon",
    73: "pizza",
    74: "platypus",
    75: "pyramid",
    76: "revolver gun",
    77: "rhinoceros",
    78: "rooster",
    79: "saxophone",
    80: "schooner sailboat",
    81: "pair of scissors",
    82: "scorpion",
    83: "sea horse",
    84: "soccer ball",
```

```
            85: "character Snoopy",
            86: "starfish",
            87: "stapler",
            88: "stegosaurus",
            89: "stop sign",
            90: "strawberry",
            91: "sunflower",
            92: "tick insect",
            93: "trilobite fossil",
            94: "umbrella",
            95: "watch",
            96: "water lily",
            97: "wheelchair",
            98: "wild cat",
            99: "Windsor chair",
            100: "wrench tool",
            101: "yin-yang symbol"
            }
        self.captions = {k: f"an image of {v}" for k, v in self.captions.items()




    def __len__(self):
        return len(self.dataset)

    def __getitem__(self, i):
        # Get the image and label from the dataset
        #print(self.dataset[i])
        #print(type(self.dataset[i]))
        #print(dir(self.dataset[i]))
        img = self.dataset[i][0]
        #print(f"img: {img.shape}")
        label = self.dataset[i][1]
        #print(f"label: {label}")

        # Apply the transformation to the image
        #img = self.transform(img)

        # Generate the caption based on the label
        caption = self.captions.get(label, "An image of an object")  # Default c
        #print(f"caption: {caption}: {len(caption)}")

        # Tokenize the caption and create the mask (use your tokenizer according
        cap, mask = tokenizer(caption, max_seq_length = max_text_seq_length)
        #print(f"tokens: {cap} {cap.shape}")
        mask = mask.repeat(len(mask), 1)
        #print(f"mask: {mask} {mask.shape}")
        #print()
```

```
        # Return the processed image, caption, and mask
        return {"image": img, "caption": cap, "mask": mask}
```

Next for convenience we define some functions to support training, the first of these functions is epoch_func, which takes a model and a data loader, this function then iterates over the data loader calculating the average loss of the model on the dataset.

```
def epoch_func(model, data_loader, optimizer = None):
    start_time = time.time()
    loss_acc = 0
    N = 0
    for i, data in enumerate(data_loader):
        img, cap, mask = data["image"].to(device), data["caption"].to(device), d
        loss = model(img,cap,mask)
        loss_acc += loss
        if optimizer is not None:
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        N += len(data["image"])
        print(f"\rBatch:{(i+1)}/{len(data_loader):.2f}, Avg Loss: {loss_acc/N:.5
        del loss, img, cap, mask

    print()
    return (loss_acc/N).detach().cpu().numpy()
```

The next helper function we define is a train_model function. this function takes a model, training data loader, epochs, learning rate, save_dir, val_loader. This function then trains for as many epochs as specified on the training set and recorder performance for both the training set and validation set for each epoch. Finally a plot is made to visualise the change in loss on both the training and validation sets at each epoch.

```
def train_model(model, train_loader, epochs, lr, save_dir, val_loader):
    os.makedirs(save_dir,  exist_ok=True)

    optimizer = optim.Adam(model.parameters(), lr=lr)

    best_loss = np.inf
    with torch.no_grad():
        initial_training_loss = epoch_func(model, train_loader, optimizer=None)
        training_losses = [initial_training_loss]

        initial_val_loss = epoch_func(model, val_loader, optimizer=None)
        validation_losses = [initial_val_loss]
    torch.cuda.empty_cache()
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_g
    num_optimized_params = sum(p.numel() for group in optimizer.param_groups for
    assert trainable_params == num_optimized_params
    print(f"Starting training {save_dir} with: {trainable_params} parameters")

    for epoch in range(epochs):
        print(f"Epoch: {epoch+1}/{epochs}")
        avg_training_loss = epoch_func(model, train_loader, optimizer=optimizer)
        training_losses.append(avg_training_loss)


        with torch.no_grad():
            avg_validation_loss = epoch_func(model, val_loader, optimizer=None)
            validation_losses.append(avg_validation_loss)

        if avg_validation_loss < best_loss:
            best_loss = avg_validation_loss
            torch.save(model.state_dict(), f"{save_dir}clip.pt")
            print("Model Saved.")
        print("\n")

    plt.plot(training_losses, label = "training")
    if val_loader is not None:
        plt.plot(validation_losses, label = "validation")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title(f"Training Loss {save_dir}")
    plt.legend()
    plt.savefig(f"{save_dir}Training.png")
    plt.show()
```

Our last helper function is called get_accuracy, as you can imagine this function takes a model, data_set and data loader and computes the accuracy of the model on that set

```
def get_accuracy(model, data_set, data_loader):
    start_time = time.time()

    # Getting dataset captions to compare images to
    text = torch.stack([tokenizer(x, max_seq_length)[0] for x in data_set.captio
    #print(f"{text.shape=}")
    mask = torch.stack([tokenizer(x, max_seq_length)[1] for x in data_set.captio
    #print(f"{mask.shape=}")

    mask = mask.repeat(1,len(mask[0])).reshape(len(mask),len(mask[0]),len(mask[0
    #print(f"{mask.shape=}")

    text_features = model.text_encoder(text, mask=mask)
    #print(f"{text_features.shape=}")
    text_features /= text_features.norm(dim=-1, keepdim=True)
    #print(f"{text_features.shape=}")

    correct, total = 0,0
    #print("\n\n")
    for i, data in enumerate(data_loader):
        images, labels = data["image"].to(device), data["caption"].to(device)
        #print(f"{images.shape=}")
        #print(f"{labels}{labels.shape}")

        image_features = model.image_encoder(images)
        #print(f"{image_features.shape=}")


        image_features /= image_features.norm(dim=-1, keepdim=True)
        #print(f"{image_features.shape=}")

        similarity = (100.0 * (image_features @ text_features.T)).softmax(dim=-1
        #print(f"{similarity.shape=}")

        _, indices = torch.max(similarity,1)
        #print(f"{indices.shape=}")
        #print("\n\n")


        pred = torch.stack([tokenizer(data_set.captions[int(i)], max_seq_length=
        #print(f"{pred.shape=}")
```

```
            #print(f"{pred==labels.shape}")
            #print(f"{torch.sum((pred==labels),dim=1)=}")
            #print(f"{torch.sum((pred==labels),dim=1)//len(pred[0])=}")
            #print(f"{sum(torch.sum((pred==labels),dim=1)//len(pred[0]))=}")
            correct += int(sum(torch.sum((pred==labels),dim=1)//len(pred[0])))
            #print(f"{correct}")
            total += len(labels)

            print(f"\rBatch:{(i+1)}/{len(data_loader)} Acc:{100 * (correct / total):
        print()

        return correct/total
```

Finally we define a function run_experiment that takes all the parameters used to define the run experiment first instatiates the model, then trains the model on the Caltech101 training set and calculates the accuracy of the learned model. Then the trained model is set to lora mode and fine tuned on the Fashion MNIST training set before the accuracy on the testing set is calculated.

```
def run_expirment(emb_dim, vit_width, img_size, patch_size, n_channels, vit_laye
                  vocab_size, text_width, max_seq_length, text_heads, text_layers,

    model = CLIP(emb_dim, vit_width, img_size, patch_size, n_channels, vit_layer

    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_g
    print(f"When training: {trainable_params} parameters")

    model.set_lora_mode(True)
    ftable_params = sum(p.numel() for p in model.parameters() if p.requires_grad
    print(f"When finetuning: {ftable_params} parameters")
    model.set_lora_mode(False)

    train_model(model, cal_train_loader, training_epochs, lr, f"{save_dir}CAL/",

    model.load_state_dict(torch.load(f"{save_dir}CAL/clip.pt", map_location=devi
    with torch.no_grad():
        cal_accuracy = get_accuracy(model, cal_test_set, cal_test_loader)
```

```
    print("\n\n\n")

    model.set_lora_mode(True)

    train_model(model, mnist_train_loader, lora_epochs, lora_lr, f"{save_dir}MNI

    model.load_state_dict(torch.load(f"{save_dir}MNIST/clip.pt", map_location=de

    with torch.no_grad():
        ft_accuracy = get_accuracy(model, mnist_test_set, mnist_test_loader)
    model.set_lora_mode(False)
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_g
    model.set_lora_mode(True)
    ftable_params = sum(p.numel() for p in model.parameters() if p.requires_grad
    with open(f"{save_dir}metrics.txt", "w") as file:
        file.write(f"trainable params: {trainable_params}\n")
        file.write(f"Caltech Accuracy: {cal_accuracy}\n")
        file.write(f"finetuneable params: {ftable_params}\n")
        file.write(f"MNIST accuracy: {ft_accuracy}\n")
    return cal_accuracy, ft_accuracy
```

Through trial and error I have found the following hyper parameters to perform well for this experiment

```
emb_dim = 32

encoder_heads=4
encoder_width=32

patch_size = (8,8)
n_channels = 1
vit_width = encoder_width
vit_heads = encoder_heads

max_seq_length = max_text_seq_length
vocab_size = 256
text_width = encoder_width
text_heads = encoder_heads

training_epochs=32
lora_epochs = 8
```

```
training_batch_size = 128
lora_batch_size = 128

encoder_layers = 4
lora_rank = 16
learning_rate = 1e-3
```

We then run our experiment with

```
cal_accuracy, ft_accuracy = run_expirment(emb_dim, vit_width, img_size, patch_si
                                                    vocab_size, text_width, ma
                                                     model_dir)
```

A complete notebook running this experiment can be found here

LoraFinetuning_CLIPModel.ipynb

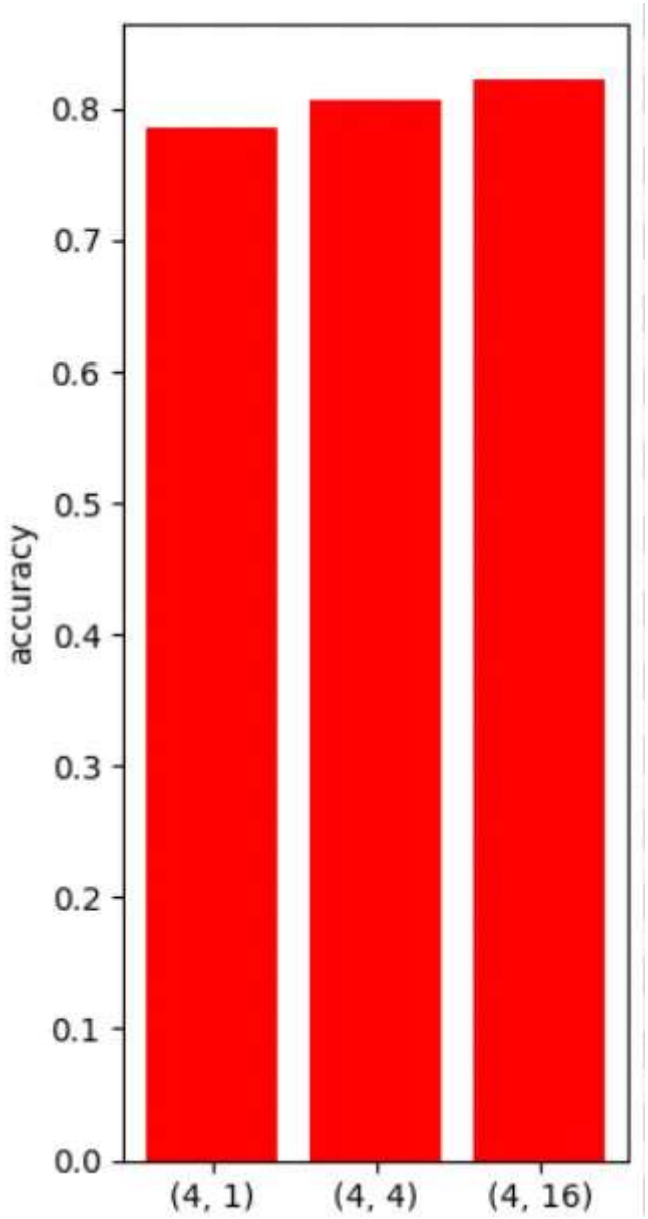
Edit description

drive.google.com

We achieve a Caltech101 accuracy of 35% when training 113,985 parameters, it is worth noting that this is significantly better than random classifier that would give 1% accuracy. This 35% accuracy can be improved with hyper parameter selection but that is outside of the scope of this article.

We achieve a Fashion MNIST accuracy of 82% while fine tuning 22,273 parameters, this is compared to the original from scratch method that achieved 84% accuracy with 65,370 parameters

We also note the impact of rank we trained models with lora ranks 1, 4, and 16 and observe that a higher rank yields better accuracy.



Accuracy for three different models with 4 encoder layers and 1, 4, and 16 rank. Accuracy only decays marginally while using only a third of the parameters.

Fine Tuning | Transformers | Machine Learning | Clip | Lora

Published in Toward Humanoids

98 Followers · Last published 3 days ago

Follow

Ongoing pre-publication research on humanoids and related technologies including paper reviews and tutorials.

Written by Maxlconway

14 Followers · 3 Following

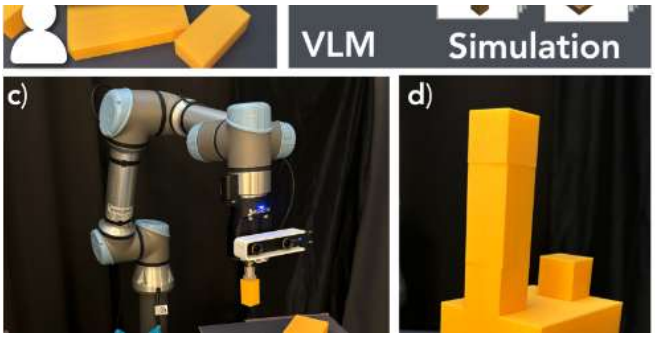
Follow

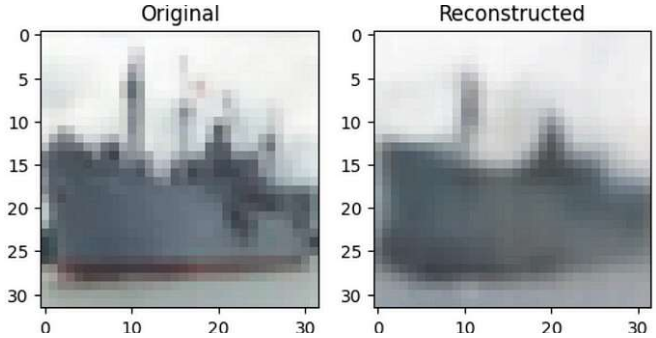
No responses yet

Write a response

What are your thoughts?

More from Maxlconway and Toward Humanoids





In Toward Humanoids by MaxIconway

Paper Review: Blox-Net, Generative Design-for-Robot-Assembly Usin...

Generating designs for robot assembly

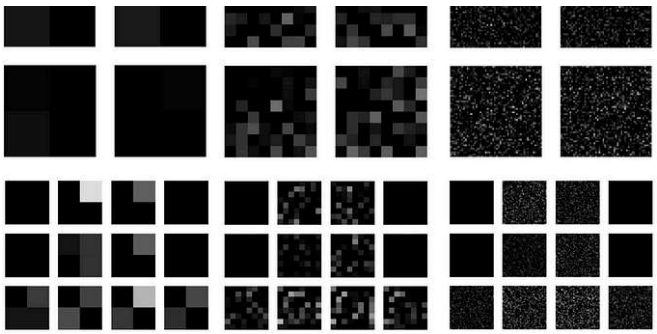
Dec 1, 2024 105

In Toward Humanoids by Nikolaus Correll

A deep-dive into Autoencoders (AE, VAE, and VQ-VAE) with code

Autoencoders thoroughly explained

Feb 18 163

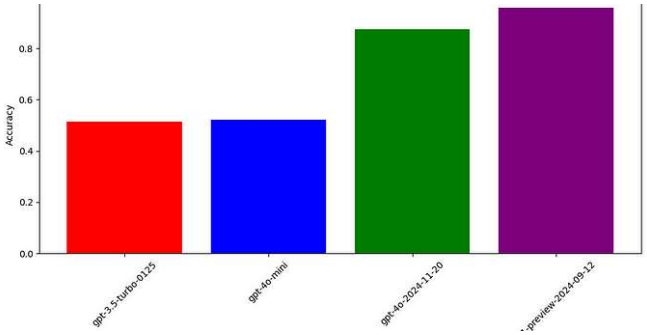


In Toward Humanoids by Nikolaus Correll

Understanding Image Patch Embeddings

From simple unfolding to 2D convolutions

Feb 4 21



In Toward Humanoids by MaxIconway

Embodied AI For Block Stacking Tasks

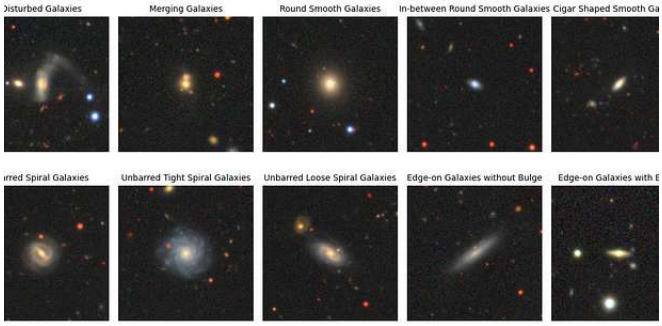
Embodied artificial intelligence is currently an active research frontier, foundation models...

Dec 17, 2024 217 9

See all from MaxIconway

See all from Toward Humanoids

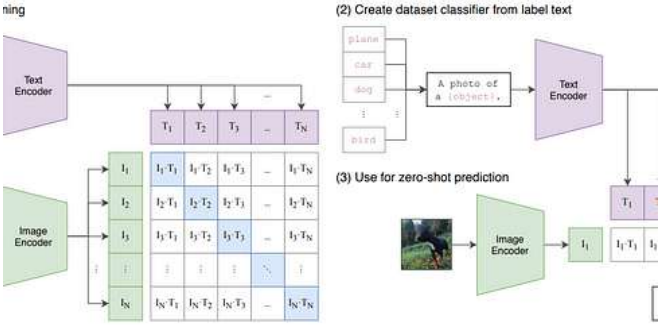
Recommended from Medium



In Toward Humanoids by Yutong Zhang

Classifying Galaxies Using CLIP Model

Oct 8, 2024 205



In Generative AI by Nick Pai

Comparison Between CLIP and BLIP Models

In recent years, vision-language models like CLIP (Contrastive Language-Image...

Nov 1, 2024 59

In Data Science Collective by Dr. Robert Kübler

Introduction to Embedding-Based Recommender Systems

Learn to build a simple recommender in TensorFlow

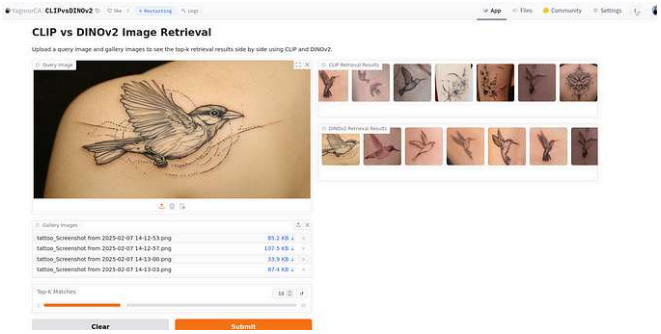
Jan 25, 2023 671 5

In Level Up Coding by Sahib Dhanjal

How To Train Your PyTorch Models With Less Memory

Strategies I regularly use to reduce GPU memory consumption by almost 20x

Feb 24 395 3



In AI Advances by Yağmur Çiğdem Aktaş

CLIP vs DINOv2: Which one is better for image retrieval?

Compare CLIP and DINOv2 for image retrieval tasks, and learn about their specific powers...

Feb 13 152

Ritvik Rastogi

Papers Explained 330: Gemini Embedding

Gemini Embedding leverages the power of Gemini to produce highly generalizable...

4d ago

See more recommendations