SambaNova SYSTEMS

LLMs on SambaNova: An Example

July 12, 2023

Safe Harbor Statement

The following is intended to outline our general product direction at this time. There is no obligation to update this presentation and the Company's products and direction are always subject to change. This presentation is intended for information purposes only and may not be relied upon for any purchasing, partnership, or other decisions.

Agenda

- Use Case
  - Model
  - Data
- Implementation
- Results & Discussion

SambaNova SYSTEMS

The Use Case

- A generative model for sentiment analysis
- We will be using a model and dataset from HuggingFace
  - This is a common use case
  - What we show here can be applied to any LLM, even those you have built yourself
- Model:
  - GPT-2 (link)
- Dataset:
  - SST2 (link)

SambaNova SYSTEMS

Training

Basic Setup

- Configure your input arguments for compilation and running
- Create dummy inputs for graph tracing
- Prepare your input data
  - Any LLM will expect certain types of input with specific shapes
  - In the case of this GPT-2 model, in addition to the raw data, it will also take in *position IDs*, *labels (input IDs shifted by 1 position)*, and *target token type IDs* that match the labels
- Create Torch DataLoaders to iterate over the input data
- Define the Optimizers
- Define the training loop
  - While not strictly necessary, you can also define an evaluation loop
  - Convert Torch Tenors to *SambaTensors*
- Convert the model graph from Torch to Samba
- Compile the model to run on RDU
- Run the model on RDU to train
- Deploy

Input Arguments - Common

- These are arguments you pass to your model from the command line
- You define them in your code and pass them to SambaFlow
- These arguments are used during compilation and/or running

```
def add_common_args(parser: argparse.ArgumentParser):
    parser.add_argument('--model_name_or_path',
                        type=str,
                        help='Path to pretrained model or model identifier from huggingface.co/models')
    parser.add_argument('--config_name',
                        type=str,
                        help='Path to pretrained model config or model identifier from huggingface.co/models')
    parser.add_argument('--cache_dir',
                        type=str,
                        help='Where to store pretrained models and data downloaded from huggingface.co')
    parser.add_argument('--max_seq_length',
                        type=int,
                        default=-1,
                        help='The maximum total input sequence length after tokenization. '
                        'Data in your data dir will be truncated or padded to this length. ')
    parser.add_argument('--weight_decay',
                        type=float,
                        default=0.1,
                        help='The weight decay to apply (if not zero) to all layers except all '
                        'bias and LayerNorm weights in the AdamW optimizer.')
    parser.add_argument('--max_grad_norm_clip',
                        type=float,
                        default=1.0,
                        help='Maximum gradient norm (for gradient clipping)')
    parser.add_argument('--learning_rate',
                        type=float,
                        default=7.5e-6,
                        help='The initial learning rate for the AdamW optimizer.')
    parser.add_argument('--dropout',
                        type=float,
                        default=0.1,
                        help='proportion of activations to drop in dropout layers')
    parser.add_argument('--prompt_loss_weight',
                        type=float,
                        default=0.0,
                        help='Relative weight of tokens with the "prompt" token type ID '
                        'during backpropagation.')
```

SambaNova SYSTEMS

Input Arguments - Running

- These arguments are used for training the model

```
def add_run_args(parser: argparse.ArgumentParser):
    parser.add_argument('--data_dir', type=str, help='Path to a directory containing HDF5 files of pre-tokenized text')
    parser.add_argument('--steps', type=int, default=800, help='Number of training steps to take')
    parser.add_argument('--min_eval_acc',
                        type=float,
                        default=0.0,
                        help='Minimum threshold for evaluation accuracy of a trained model. only for testing.')
    parser.add_argument('--subsample_eval',
                        type=float,
                        default=0.1,
                        help='Proportion of the evaluation set to use for evaluation. '
                        'Setting a smaller poportion helps speed up evauation.')
    parser.add_argument('--checkpoint_name',
                        type=str,
                        default='checkpoint.pt',
                        help='Path where the final trained checkpoint will be saved.')
```

Dummy Inputs for Compilation

- In order for the SambaFlow compiler to map the model graph onto an RDU, it must trace how the model's input tensors change shape to produce the final output tensors
- This doesn't require actual data, only tensors of the same shape
- Note the call to **samba.from_torch_tensor()**
  - The RDU manipulates **SambaTensors**, not Torch Tensors

```python
def get_compile_inputs(args: argparse.Namespace) -> Tuple[Any]:
    """
    Generate compile-time tracing inputs to trace the model graph.
    Since they're only used for tracing, these tensors are composed of dummy data, and only have
    the shapes of the corresponding tensors used in the real forward pass.
    """
    # Make input_ids
    input_ids = torch.randint(0, 5000, (args.batch_size, args.max_seq_length), dtype=torch.int32)
    input_ids = samba.from_torch_tensor(input_ids, name="input_ids")

    # Make position_ids
    position_ids = torch.arange(args.max_seq_length)
    position_ids = position_ids.short()
    position_ids = samba.from_torch_tensor(position_ids.unsqueeze(0).expand(input_ids.shape), name='input_position_ids')

    # Make labels
    labels = torch.ones(args.batch_size, args.max_seq_length, dtype=torch.int16)
    labels = samba.from_torch_tensor(labels, name='labels')

    # Prepare the tracing items
    tracing_inputs = (input_ids, None, None, None, position_ids, None, None, None, None, labels)
    return tracing_inputs
```

Prepare Inputs

- This creates:
  - **Position IDs**
  - **Labels**
  - **Target Token Type IDs**
- These tensors aren't part of the dataset, but are needed by the model
  - Their shape and values will be dependent on the dataset, though
- The output tuple of tensors of this function should be the same shape as the dummy tensors used during tracing

```
def prepare_inputs(args: argparse.Namespace,
                   inputs: Tuple[torch.Tensor, torch.Tensor]) -> Tuple[Sequence[Optional[torch.Tensor]], torch.Tensor]:
    """
    Prepare a batch of torch tensors from the dataloader for passing into the model.
    This involves creating position IDs, shifting over the input_ids by 1 position
    to create labels, and creating target_token_type_ids to match the labels. The final
    tuple of tensors should match the shape of the inputs used for tracing.
    """
    input_ids = inputs[0].int()
    batch_size = input_ids.shape[0]

    # Train dataloader does not contain the following entries
    position_ids = torch.arange(args.max_seq_length)

    # Prepare the 3D attention mask for the Huggingface Module
    # set the labels to the input_ids, to be modified in the GPT model
    labels = input_ids.short()
    labels = labels[..., 1:]
    labels = torch.cat((labels, torch.ones([labels.shape[0], 1], dtype=labels.dtype) * -100), dim=1)

    position_ids = position_ids.unsqueeze(0).expand(input_ids.shape)

    position_ids = position_ids.short()

    token_type_ids = inputs[1].int()
    target_token_type_ids = token_type_ids[..., 1:]
    target_token_type_ids = torch.cat(
        (target_token_type_ids,
         torch.ones([target_token_type_ids.shape[0], 1], dtype=target_token_type_ids.dtype) * PADDING_TOKEN_TYPE_ID),
        dim=1)

    if batch_size < args.batch_size:
        # isn't necessary since dataloader drop_last=True
        input_ids = pad_tensor(input_ids, args.batch_size, 0)
        position_ids = pad_tensor(position_ids, args.batch_size, 0)
        labels = pad_tensor(labels, args.batch_size, -100)
        target_token_type_ids = pad_tensor(target_token_type_ids, args.batch_size, PADDING_TOKEN_TYPE_ID)
    traced_inputs = (input_ids, None, None, None, position_ids, None, None, None, None, labels)

    return traced_inputs, target_token_type_ids
```

SambaNova SYSTEMS

Prepare Inputs - What are SambaTensors?

- SambaFlow works with **SambaTensors**, which are a wrapper around Torch Tensors
  - Each has a unique **name** for tracing as well as a **batch_dim** for optimization
- Conversion from Torch Tensors to SambaTensors is easy:
  - **samba.from_torch_tensor(<torch_tensor>, name="<x>")**

```
def get_runtime_inputs(torch_input: Sequence[Optional[samba.SambaTensor]]):
    """
    Given a batch from the torch dataloader, transform them into SambaTensors ready for session.run to consume.
    """
    torch_input = torch_input if len(torch_input) == 4 else ([torch_input[0]] + [None] + torch_input[1:])
    input_ids, attention_mask, position_ids, labels = torch_input
    input_ids = samba.from_torch_tensor(input_ids.int(), name="input_ids")
    position_ids = samba.from_torch_tensor(position_ids, name='input_position_ids')
    labels = samba.from_torch_tensor(labels, name='labels')
    if attention_mask is not None:
        attention_mask = samba.from_torch_tensor(attention_mask, batch_dim=0, name="attention_mask")
        return [input_ids, attention_mask, position_ids, labels]
    else:
        return [input_ids, position_ids, labels]
```

SambaNova SYSTEMS

Data Loading - Training

- Create a set of DataLoaders over the training data files, one DataLoader per file
- We provide a wrapper, **PretrainingGenerativeDataset**, that creates a Torch Dataset object for each data file
- By setting **drop_last=True** in the DataLoader, we avoid having to pad the batch dimension

```
def get_epoch_train_iterators(args):
    """
    Get a list of dataloaders that will iterate over all of the files in the training dataset
    """
    files = [
        os.path.join(args.data_dir, f) for f in os.listdir(args.data_dir)
        if os.path.isfile(os.path.join(args.data_dir, f)) and ('train' in f)
    ]
    files.sort()
    len(files)
    dataloaders = []
    for data_file in files:
        train_data = PretrainingGenerativeDataset(input_file=data_file)
        train_sampler = RandomSampler(train_data)
        train_dataloader = DataLoader(train_data, sampler=train_sampler, batch_size=args.batch_size, drop_last=True)

        dataloaders.append(train_dataloader)
    return dataloaders
```

SambaNova SYSTEMS

Data Loading - Evaluation

- Create a set of DataLoaders over the training data files, one DataLoader per file
- We provide a wrapper, **PretrainingGenerativeDataset**, that creates a Torch Dataset object for each data file
- By setting **drop_last=True** in the DataLoader, we avoid having to pad the batch dimension

```python
def get_eval_iterators(args):
    """
    Get a list of dataloaders that will iterate over all of the files in the eval dataset
    """
    files = [
        os.path.join(args.data_dir, f) for f in os.listdir(args.data_dir)
        if os.path.isfile(os.path.join(args.data_dir, f)) and ('dev' in f or 'test' in f)
    ]
    files.sort()
    num_files = len(files)

    assert 0.0 <= args.subsample_eval <= 1.0, "Subsample eval should be between [0, 1.0]"
    # Subsample the validation file
    num_files_to_evaluate = int(math.ceil(args.subsample_eval * num_files))
    assert num_files_to_evaluate > 0, "Must have at least 1 eval file! " + \
        "Try increasing args.subsample_eval to a large value (max 1.0) or " + \
        "check the file dir to see if the files are missing"

    files_to_eval = random.sample(range(num_files), k=num_files_to_evaluate)

    dataloaders = []
    for k, data_file in enumerate(files):
        if k not in files_to_eval:
            continue
        eval_data = PretrainingGenerativeDataset(input_file=data_file)
        eval_sampler = SequentialSampler(eval_data)
        eval_dataloader = DataLoader(eval_data, sampler=eval_sampler, batch_size=args.batch_size, drop_last=True)
        dataloaders.append(eval_dataloader)

    num_samples = sum([len(dataloader) for dataloader in dataloaders])
    print(
        f"Evaluating on {num_files_to_evaluate} files out of {num_files} with {num_samples} total samples in the evaluation dataset",
        flush=True)

    return dataloaders
```

SambaNova SYSTEMS

Data Loading - Creating Torch Datasets

- We add an additional field to the dataset: **token_type_ids**

```
class PretrainingGenerativeDataset(torch.utils.data.Dataset):
    """
    Torch Dataset for a file uesd in generative tuning
    """
    def __init__(self, input_file):
        self.input_file = input_file
        f = h5py.File(input_file, "r")
        # extra field 'token_type_ids'
        keys = ['input_ids', 'token_type_ids']
        self.inputs = [np.asarray(f[key][:]) for key in keys]

        f.close()

    def __len__(self):
        'Denotes the total number of samples'
        return len(self.inputs[0])

    def __getitem__(self, index):
        [input_ids, token_type_ids] = [torch.from_numpy(input[index].astype(np.int32)) for input in self.inputs]
        return [input_ids.long(), token_type_ids.long()]
```

SambaNova SYSTEMS

Define Optimizers

- SambaFlow has built-in optimizers
- We'll use **AdamW**
  - It provides good convergence for Transformer models

```
def get_optimizers(args: argparse.Namespace, model: torch.nn.Module):
    """
    Construct the optimizers
    """
    emb_modules = [module for module in model.modules() if isinstance(module, torch.nn.Embedding)]
    emb_params = OrderedSet(itertools.chain(*[emb.parameters() for emb in emb_modules]))
    other_params = OrderedSet([(name, param) for name, param in model.named_parameters() if param not in emb_params])

    # Exclude weight decay from bias & layernorm parameters
    no_decay = ["bias"]
    for name, params in model.named_parameters():
        if "ln" in name or "layernorm" in name or "layer_norm" in name:
            no_decay.append(name)
    params_w_weight_decay = OrderedSet([(n, p) for n, p in other_params if not any(nd in n for nd in no_decay)])
    params_wo_weight_decay = OrderedSet([(n, p) for n, p in other_params if any(nd in n for nd in no_decay)])

    emb_optim = samba.optim.AdamW(emb_params,
                                  lr=args.learning_rate,
                                  betas=(0.9, 0.997),
                                  eps=1e-8,
                                  weight_decay=args.weight_decay,
                                  max_grad_norm=args.max_grad_norm_clip)
    opt_w_weight_decay = samba.optim.AdamW([param for (name, param) in params_w_weight_decay],
                                           lr=args.learning_rate,
                                           betas=(0.9, 0.997),
                                           weight_decay=args.weight_decay,
                                           max_grad_norm=args.max_grad_norm_clip)
    opt_wo_weight_decay = samba.optim.AdamW([param for (name, param) in params_wo_weight_decay],
                                            lr=args.learning_rate,
                                            betas=(0.9, 0.997),
                                            weight_decay=0,
                                            max_grad_norm=args.max_grad_norm_clip)

    return [emb_optim, opt_w_weight_decay, opt_wo_weight_decay]
```

SambaNova SYSTEMS

Single Model Step

- Scale loss depending on the labels indicating padding tokens/ignored indices during this step
  - Use to initialize RDU gradient tensor for this step
- To run, call **samba.session.run()**:
  - Pass in input tensors
  - Pass in the traced outputs (initially from the compilation)
  - The **hyperparam_dict** allows one to pass in values that can change during runtime, e.g., the LR
  - Specify which sections to run - an RDU can run the Forward, Backward, Gradient and Optimizer passes simultaneously
- Return the loss
  - Convert back to Torch Tensor for later evaluation on CPU
  - Apply loss scale factor and sum over the tensor to get final value

```
def model_step(args, model, inputs, target_token_type_ids, traced_outputs):
    """
    Take one training step on RDU
    """
    inputs = [ipt for ipt in inputs if ipt is not None]
    learning_rate = args.learning_rate
    dropout_rate = args.dropout
    hyper_dict = {'lr': learning_rate}
    dropout_dict = {'p': dropout_rate}

    hyperparam_dict = {**hyper_dict, **dropout_dict}

    # Compute loss scale
    loss_scale = compute_loss_scale(args, inputs[-1], target_token_type_ids, model.output_tensors[0].dtype)

    # Prepare inputs
    grad_of_outputs_to_run = []
    inputs_to_run = []

    inputs_this_step = get_runtime_inputs(inputs)

    inputs_to_run.append(inputs_this_step)
    grad_of_outputs_to_run.append(loss_scale)

    traced_outputs[0].sn_grad = loss_scale
    outputs = samba.session.run(inputs_this_step,
                                traced_outputs,
                                hyperparam_dict=hyperparam_dict,
                                section_types=['FWD', 'BCKWD', 'GRADNORM', 'OPT'])

    samba_loss = outputs[0]
    loss = samba.to_torch(samba_loss).float()
    loss *= loss_scale.float()
    loss = loss.sum()
    return loss
```

SambaNova SYSTEMS

Training Loop

- The outermost loop checks for completion, based on the number of given training steps
  - It gets the training DataLoaders: **get_epoch_train_iterators()**
  - It's here, after training completes, that evaluation happens: **evaluate()**
- The innermost training loop iterates over batches from the current DataLoader from the next innermost loop
  - It prepares the inputs for processing: **prepare_inputs()**
  - Then does 1 step of training on RDU: **model_step()**
- Upon completion, a checkpoint is created: **save_checkpoint()**

```
def train(args, model, traced_outputs):
    """
    Perform the training procedure
    """
    # Create tokenizer
    if args.model_name_or_path is None:
        raise ValueError("Provide a --model_name_or_path for the tokenizer! e.x 'gpt2'")
    tokenizer = AutoTokenizer.from_pretrained(args.model_name_or_path, cache_dir=args.cache_dir)

    eval_total_loss = None
    epoch = 1
    total_steps_taken = 0
    training_done = False

    # Depend on the provided step count rather than epochs
    while not training_done:
        print(f"Training Epoch {epoch}")
        dataloaders = get_epoch_train_iterators(args)
        with build_progress_bar(dataloaders) as pbar:

            for dataloader in dataloaders:

                if total_steps_taken >= args.steps:
                    if not training_done:
                        print(f"Finished training at {total_steps_taken} steps!")
                    training_done = True
                    break

                for batch in dataloader:

                    # Break out if steps exceed specified steps
                    if total_steps_taken >= args.steps:
                        if not training_done:
                            print(f"Finished training at {total_steps_taken} steps!")
                        training_done = True
                        break

                    inputs, target_token_type_ids = prepare_inputs(args, batch)
                    inputs = [t for t in inputs if t is not None]

                    # Take one training step
                    loss = model_step(args, model, inputs, target_token_type_ids, traced_outputs)
                    train_loss = loss.item()
                    total_steps_taken += 1
                    pbar.update(1)
                    pbar.set_description(f"Training loss: {train_loss}")

        if not training_done:
            eval_total_loss, eval_acc = evaluate(args, model, traced_outputs)
            print(
                f"Evaluation Results At Step {total_steps_taken} : Total Loss: {eval_total_loss}, Eval acc: {eval_acc}")
            epoch += 1
    print("Finished training")
    # Eval
    eval_total_loss, eval_acc = evaluate(args, model, traced_outputs)

    print(f"Final eval total loss: {eval_total_loss}\nFinal eval accuracy: {eval_acc}")

    if args.min_eval_acc > 0.0:
        assert eval_acc >= args.min_eval_acc, f"Obtained eval_acc={eval_acc}, Expected Minimum eval_acc={args.min_eval_acc}"

    # Save checkpoint
    # move checkpoint values back to host
    save_checkpoint(model, args.checkpoint_name)
```

Evaluation

- Get the evaluation DataLoaders: **get_eval_iterators()**
- The outermost loop iterates over the DataLoaders
- The next inner loop iterates over batches from the current DataLoader:
  - Prepares inputs: **prepare_input()**
  - Converts the tensors to SambaTensors: **get_runtime_inputs()**
  - Runs only the forward pass: **samba.session.run()**
  - Saves the logits
- The innermost loop iterates over each sample in the current batch:
  - It gets the actual targets for the sample
  - Uses the sample's logit to predict labels
  - Samples that have no completion token are ignored
  - Saves the index of target token type ID of the sample if it matches the completion token type ID
  - Save target label and the prediction at that index
- As in the single model step, we compute the loss scale to the loss from **run()**
- Compute total eval loss and final eval accuracy: **exact_match_accuracy()**

```python
def evaluate(args, model, traced_outputs):
    """
    Evaluate the model's performance on RDU
    """
    total_eval_loss = []
    labels_list = []
    preds_list = []

    eval_dataloaders = get_eval_iterators(args)

    with build_progress_bar(eval_dataloaders) as pbar:
        for eval_iter in eval_dataloaders:
            for step, batch in enumerate(eval_iter):
                inputs, target_token_type_ids = prepare_inputs(args, batch)

                inputs = [t for t in inputs if t is not None]
                hyperparam_dict = {}
                hyperparam_dict['lr'] = 0.0
                hyperparam_dict['p'] = 0.0

                # prepare current step input & perform model fwd
                inputs_this_step = get_runtime_inputs(inputs)
                outputs = samba.session.run(inputs_this_step,
                                            traced_outputs,
                                            hyperparam_dict=hyperparam_dict,
                                            section_types=["FWD"])
                logits = samba.to_torch(outputs[1])
                # Compute eval acc
                for sample in range(args.batch_size):
                    targets = inputs_this_step[-1][sample]
                    logit = logits[sample]
                    preds = torch.argmax(logit, axis=-1)
                    # if no completion tokens, it's most likely padded samples at the end of the last batch
                    # we exclude these samples from metric calculation
                    if COMPLETION_TOKEN_TYPE_ID not in target_token_type_ids[sample]:
                        continue
                    idx = (target_token_type_ids[sample] == COMPLETION_TOKEN_TYPE_ID)
                    # Make sure label and pred have the same dtype
                    labels_list.append(samba.to_torch(targets[idx]).short())
                    preds_list.append(samba.to_torch(preds[idx]).short())

                # Compute loss scale
                loss_scale = compute_loss_scale(args, inputs[-1], target_token_type_ids, model.output_tensors[0].dtype)
                samba_loss = outputs[0]
                loss = samba.to_torch(samba_loss).float()
                loss *= loss_scale.float()
                loss = loss.sum()
                loss = samba.to_torch(loss)
                total_eval_loss.append(loss.item())
                pbar.update(1)

    # Compute exact match acc
    eval_acc = exact_match_accuracy(labels_list, preds_list)
    return sum(total_eval_loss) / len(total_eval_loss), eval_acc
```

SambaNova SYSTEMS

Tying it all together

- Bring in all of the args: **parse_app_args()**
- We use a pretrained GPT-2 model from HF:
  - We can pull the model and its config from HF:
    - **AutomodelForCausalLM.from_pretrained()**
  - Or, we can provide our own config file to pass to HF:
    - **AutoConfig.from_pretrained()**
    - **AutoModelForCausalLM.from_config()**
- Train the HF model
- Patch the model to improve RDU performance: **patch_model()**
- Convert model to Samba: **samba.from_torch_model_()**
- Get dummy inputs: **get_compile_inputs()**
- Get optimizers: **get_optimizers()**
- Compile the model with: **samba.session.compile()**
- Train the model on RDU
  - **utils.trace_graph()**
  - **train()**

```python
def main(argv: List[str]) -> None:
    # Parse the args
    args = parse_app_args(argv=argv, common_parser_fn=add_common_args, run_parser_fn=add_run_args)

    # Download the model from HuggingFace
    if args.model_name_or_path:
        model = AutoModelForCausalLM.from_pretrained(args.model_name_or_path, cache_dir=args.cache_dir)
        model.training = True
    elif args.config_name:
        config = AutoConfig.from_pretrained(args.config_name, cache_dir=args.cache_dir)
        # Read dropout rate from config
        args.dropout = config.resid_pdrop
        model = AutoModelForCausalLM.from_config(config)
    else:
        raise RuntimeError("Must provide --model_name_or_path or --config_name")

    if not args.inference:
        model = model.train()
    else:
        model = model.eval()

    # Patch the model here
    model = patch_model(model, args)

    samba.from_torch_model_(model)

    # Make the tracing inputs
    inputs = get_compile_inputs(args)

    # Make the optimizer
    optims = get_optimizers(args, model)

    if args.command == 'compile':
        print(f"DGDG:{args.inference}")
        samba.session.compile(model,
                              inputs,
                              optims,
                              name='hf_transformer',
                              app_dir=samba.utils.get_file_dir(__file__),
                              init_output_grads=True,
                              inference=args.inference)
    elif args.command == 'run':
        traced_outputs = utils.trace_graph(model, inputs, optims, pef=args.pef, init_output_grads=True)
        train(args, model, traced_outputs)
```

SambaNova SYSTEMS

Patching a Model

- Patching an HF model can improve overall performance on RDU
- Not every model will need patching

```python
def patch_model(model, args):
    """
    Patch the HuggingFace model to facilitate compilation and running on RDU
    """
    model.return_logits = True
    model.return_cache = False
    model.no_index_select_patch = True
    return gpt2_patch_helper(model,
                             args.batch_size,
                             inference=args.inference,
                             max_length=args.max_seq_length,
                             max_pef_length=args.max_seq_length)
```

```python
def gpt2_patch_helper(model: nn.Module,
                      batch_size: int,
                      inference: bool = False,
                      max_length: int = 2048,
                      max_pef_length: int = 2048):

    model.forward = gpt2_head_forward.__get__(model, transformers.models.gpt2.modeling_gpt2.GPT2LMHeadModel)

    for ind, layers in enumerate(model.transformer.h):
        layers.mlp.act = nn.GELU()
        size = (1, 1, layers.attn.bias.shape[-1], layers.attn.bias.shape[-2])
        layers.attn.bias = layers.attn.bias.expand(size)
        layers.attn.c_attn.weight = transpose_weight(layers.attn.c_attn.weight)
        layers.attn.c_proj.weight = transpose_weight(layers.attn.c_proj.weight)
        layers.mlp.c_fc.weight = transpose_weight(layers.mlp.c_fc.weight)
        layers.mlp.c_proj.weight = transpose_weight(layers.mlp.c_proj.weight)

        layers.attn.c_attn.forward = gpt2_conv1d_forward.__get__(layers.attn.c_attn, transformers.modeling_utils.Conv1D)
        layers.attn.c_proj.forward = gpt2_conv1d_forward.__get__(layers.attn.c_proj, transformers.modeling_utils.Conv1D)
        layers.mlp.c_fc.forward = gpt2_conv1d_forward.__get__(layers.mlp.c_fc, transformers.modeling_utils.Conv1D)
        layers.mlp.c_proj.forward = gpt2_conv1d_forward.__get__(layers.mlp.c_proj, transformers.modeling_utils.Conv1D)

        # op_fusion(layers, 'Encoder')
        hyper_sec(layers, 'Encoder')

    return model
```

SambaNova SYSTEMS

Helpful Functions

- Optional, but helpful functions
  - **exact_match_accuracy()**
  - **save_checkpoint()**
  - **load_checkpoint()**

```python
def exact_match_accuracy(labels_list, preds_list):
    """ Convenience function to use a metric calculation from HF """
    assert len(labels_list) == len(preds_list)
    total = 0
    match = 0
    for (label, pred) in zip(labels_list, preds_list):
        total += 1
        if torch.equal(label, pred):
            match += 1
    return 1.0 * match / total
```

```python
def save_checkpoint(model, checkpoint_dir):
    """
    Transfer model weights from RDU to Host and save them in a PyTorch Checkpoint
    """
    samba.session.to_cpu(model)

    state_dict = model.state_dict()
    # Save each tensor as a torch Tensor rather than a SambaTensor for portability
    for key, val in state_dict.items():
        if isinstance(val, samba.SambaTensor):
            state_dict[key] = val.torch_tensor()
    print(f"Saving Checkpoint to disk at {checkpoint_dir}")
    torch.save(state_dict, checkpoint_dir)


def load_checkpoint(model, checkpoint_dir):
    """
    Load model weights from a checkpoint on Host and transfer them to RDU
    """
    print(f"Loading Checkpoint from disk at {checkpoint_dir}")
    checkpoint_state_dict = torch.load(checkpoint_dir)
    model.load_state_dict(checkpoint_state_dict)
    samba.session.to_rdu(model)
```

SambaNova SYSTEMS

Compile & Training Commands

Compile Command

```
SN_NUM_THREADS=32 python tutorial_train.py compile \
--max_seq_length 1024 \
-b 16 \
--config_name <path/to/config.json> \
--weight_decay 0.1 \
--max_grad_norm_clip 1.0 \
--model_name_or_path gpt2 \
--num_tiles 4 \
--pef <name_of_pef>
```

Training Command

```
SN_NUM_THREADS=32 python tutorial_train.py run \
--max_seq_length 1024 \
-b 16 \
--weight_decay 0.1 \
--max_grad_norm_clip 1.0 \
--data_dir <path/to/dataset> \
--checkpoint_name <name_of_checkpoint> \
--model_name_or_path gpt2 \
--steps 800 \
--num_tiles 4 \
--min_eval_acc 0.87 \
--pef <path/to/pef>
```

Configuration File


```
{
  "activation_function": "gelu_new",
  "architectures": [
    "GPT2LMHeadModel"
  ],
  "attn_pdrop": 0.1,
  "bos_token_id": 50256,
  "embd_pdrop": 0.1,
  "eos_token_id": 50256,
  "initializer_range": 0.02,
  "layer_norm_epsilon": 1e-05,
  "model_type": "gpt2",
  "n_ctx": 1024,
  "n_embd": 768,
  "n_head": 12,
  "n_layer": 12,
  "n_positions": 1024,
  "resid_pdrop": 0.1,
  "summary_activation": null,
  "summary_first_dropout": 0.1,
  "summary_proj_to_labels": true,
  "summary_type": "cls_index",
  "summary_use_proj": true,
  "task_specific_params": {
    "text-generation": {
      "do_sample": true,
      "max_length": 50
    }
  },
  "vocab_size": 50257
}
```


SambaNova SYSTEMS

SambaNova SYSTEMS

Generation

Basic Setup for Generation

- It's purposely very similar to the training setup
- Configure you input arguments for compilation and running
- Create dummy inputs for graph tracing
- This is where the generative model differs:
  - It is an **inference** model, so there is no input data to train with
    - We don't need DataLoaders, optimizers or a training loop
  - It still needs to be compiled, but we will compile for inference only
  - Weights will be loaded from the checkpoint previously created by the training model
  - The prompts that will be used to make predictions will need processing
- Convert the model graph from Torch to Samba
- Compile the model to run inference on RDU
- Run the model on RDU to generate predictions

Input Arguments - Common & Running

- These are arguments you pass to your model from the command line
- You define them in your code and pass them to SambaFlow
- These arguments are used during compilation and running

```python
def add_common_args(parser: argparse.ArgumentParser):
    parser.add_argument('--model_name_or_path',
                        type=str,
                        help='Path to pretrained model or model identifier from huggingface.co/models')
    parser.add_argument('--config_name',
                        type=str,
                        help='Path to pretrained model config or model identifier from huggingface.co/models')
    parser.add_argument('--cache_dir',
                        type=str,
                        help='Where to store pretrained models and data downloaded from huggingface.co')
    parser.add_argument('--max_seq_length',
                        type=int,
                        default=-1,
                        help='The maximum total input sequence length after tokenization. '
                        'Data in your data dir will be truncated or padded to this length. ')
    parser.add_argument('--examples_to_generate',
                        type=int,
                        default=20,
                        help='The number of prompts to run generation on')


def add_run_args(parser: argparse.ArgumentParser):
    parser.add_argument(
        '--data_dir',
        type=str,
        help='Path to a .json file, .jsonl file or a directory containing .jsonl files. '
        'Each json object should contain a "prompt" key of text used for prompting model text generation.')
    parser.add_argument('--max_tokens_to_generate',
                        default=20,
                        type=int,
                        help='Maximum tokens to generate after each prompt.')
    parser.add_argument(
        '--checkpoint_name',
        type=str,
        default='',
        help='Path to a checkpoint containing weight with names matching those provided by the --model_name_or_path')
```

SambaNova SYSTEMS

Dummy Inputs for Compilation

- In order for the SambaFlow compiler to map the model graph onto an RDU, it must trace how the model's input tensors change shape to produce the final output tensors
- This doesn't require actual data, only tensors of the same shape
- Note the call to **samba.from_torch_tensor()**
  - The RDU manipulates **SambaTensors**, not Torch Tensors

```python
def get_compile_inputs(args: argparse.Namespace) -> Sequence[Optional[samba.SambaTensor]]:
    """
    Get input tensors to use for tracing the model.
    Since they're only used for tracing, these tensors are composed of dummy data, and only have
    the shapes of the corresponding tensors used in the real forward pass.
    """
    batch_size = args.batch_size
    length = args.max_seq_length

    assert batch_size == 1, "Only batch size 1 is supported at the moment"

    # Input IDs
    input_ids = torch.randint(0, 5000, (batch_size, length)).int()
    input_ids = samba.from_torch_tensor(input_ids, name='input_ids')

    # Position IDs
    position_ids = torch.arange(length)
    position_ids = position_ids.short()
    position_ids = samba.from_torch_tensor(position_ids.unsqueeze(0).expand(input_ids.shape), name='input_position_ids')

    # Attention Mask
    # Prepare the 3D attention mask for the Huggingface Module
    attention_mask = torch.randint(2, (batch_size, length), dtype=torch.bfloat16)
    attention_mask = attention_mask[:, None, :].to(torch.float32)
    attention_mask_name = 'attention_mask'
    attention_mask = samba.from_torch_tensor(attention_mask, name=attention_mask_name)

    past_key_values = None

    traced_inputs = (input_ids, past_key_values, attention_mask, None, position_ids, None, None, None)

    return traced_inputs
```

SambaNova SYSTEMS

Prepare Inputs

- This creates:
  - **Input IDs**
  - **Attention Mask**
  - **Position IDs**
- Tensors are converted to SambaTensors

```python
def get_runtime_inputs(inputs: Dict[str, List[Any]], max_seq_length: int):
    """
    Given inputs from the dataset, create inputs for samba.session.run
    """

    # Create input_ids
    input_ids = inputs['input_ids']

    # Pad the inputs to the appropriate max sequence length
    input_ids = F.pad(input_ids, (0, max_seq_length - input_ids.shape[1]))
    input_ids = samba.from_torch_tensor(input_ids.int(), name="input_ids")

    # Create attention_mask
    attention_mask = inputs['attention_mask']
    attention_mask = F.pad(attention_mask, (0, max_seq_length - attention_mask.shape[1]))
    attention_mask = attention_mask[:, None, :].to(torch.float32)
    attention_mask = samba.from_torch_tensor(attention_mask, name='attention_mask')

    # Create position_ids
    position_ids_torch = torch.arange(max_seq_length).short()
    position_ids = samba.from_torch_tensor(position_ids_torch.unsqueeze(0).expand(input_ids.shape),
                                           name='input_position_ids')

    traced_inputs = (input_ids, None, attention_mask, None, position_ids, None, None, None)

    return traced_inputs
```

SambaNova SYSTEMS

Generate Predictions

- Load the checkpoint
- Define the single model step
  - Unlike previous version, this one handles preparing its inputs: **get_runtime_inputs()**
  - As this is inference, **samba.session.run()** only needs to run the forward pass to get the logits
  - This function will be called in place of the model's original forward()
  - So, it returns a **CausalLMOuputWithCrossAttentions()** object
- Use the GPT-2 tokenizer
- The prompt file is in .jsonl format, **GenerativeDataset()** converts it to a Torch Dataset
- Finally, iterate over the dataset, tokenizing the prompts
  - Generate predictions via the model's HF **generate()** function
  - Decode the generated tokens into text

```python
def generate(args, model, traced_outputs):
    """
    Generate some outputs from the model, hooking into the Huggingface generate function
    """

    # Load the checkpoint
    if args.checkpoint_name:
        load_checkpoint(model, args.checkpoint_name)

    # Define the internal logits fwd pass in terms of session.run
    def model_rdu_step(self, *input, **kwargs):
        input_id_length = kwargs['input_ids'].shape[1]
        samba_inputs = get_runtime_inputs(kwargs, args.max_seq_length)

        output_logits = samba.session.run(input_tensors=samba_inputs,
                                          output_tensors=traced_outputs,
                                          hyperparam_dict={'p': 0.0},
                                          section_types=['fwd'])[0]
        logits = samba.to_torch(output_logits)[:, :input_id_length, :].float()
        return CausalLMOutputWithCrossAttentions(loss=None, logits=logits)

    # Replace the model's internal forward call with the RDU step so it is automatically called during generate
    base_model_class = model.__class__
    base_model_class.__torch_call__ = base_model_class.__call__
    base_model_class.__call__ = model_rdu_step

    # Make a tokenizer, model checkpoint folder has tokenizer info vocab.json and merges.txt
    tokenizer = AutoTokenizer.from_pretrained(args.model_name_or_path, cache_dir=args.cache_dir)

    # Make a dataset
    dataset = GenerativeDataset(args.data_dir)
    predictions = []
    # Generate the stuff

    for k, example in enumerate(dataset):
        if k >= args.examples_to_generate:
            break
        # Tokenize inputs
        model_inputs = tokenizer(example['prompt'], return_tensors='pt')
        input_ids = model_inputs['input_ids']
        input_length = input_ids.shape[-1]

        # Hook into HF model.generate to generate predictions. The above __call__ patching will ensure the model runs on RDU
        generated_ids = model.generate(model_inputs['input_ids'], max_length=input_length + args.max_tokens_to_generate)
        generated_text = tokenizer.decode(generated_ids.squeeze(0))
        predictions.append(generated_text)

    return predictions
```

SambaNova SYSTEMS

Patching a Model

- Patching an HF model can improve overall performance on RDU
- Not every model will need patching
- This patch is very similar to the one previously used

```python
def patch_model(model, args):
    """
    Patch the HuggingFace model to facilitate compilation
    """
    model.return_logits = True
    model.no_index_select_patch = True
    return gpt2_patch_helper(model,
                             args.batch_size,
                             inference=args.inference,
                             max_length=args.max_seq_length,
                             max_pef_length=args.max_seq_length)
```

```python
def gpt2_patch_helper(model: nn.Module,
                      batch_size: int,
                      inference: bool = False,
                      max_length: int = 2048,
                      max_pef_length: int = 2048):

    model.forward = gpt2_head_forward.__get__(model, transformers.models.gpt2.modeling_gpt2.GPT2LMHeadModel)

    for ind, layers in enumerate(model.transformer.h):
        layers.mlp.act = nn.GELU()
        size = (1, 1, layers.attn.bias.shape[-1], layers.attn.bias.shape[-2])
        layers.attn.bias = layers.attn.bias.expand(size)
        layers.attn.c_attn.weight = transpose_weight(layers.attn.c_attn.weight)
        layers.attn.c_proj.weight = transpose_weight(layers.attn.c_proj.weight)
        layers.mlp.c_fc.weight = transpose_weight(layers.mlp.c_fc.weight)
        layers.mlp.c_proj.weight = transpose_weight(layers.mlp.c_proj.weight)

        layers.attn.c_attn.forward = gpt2_conv1d_forward.__get__(layers.attn.c_attn, transformers.modeling_utils.Conv1D)
        layers.attn.c_proj.forward = gpt2_conv1d_forward.__get__(layers.attn.c_proj, transformers.modeling_utils.Conv1D)
        layers.mlp.c_fc.forward = gpt2_conv1d_forward.__get__(layers.mlp.c_fc, transformers.modeling_utils.Conv1D)
        layers.mlp.c_proj.forward = gpt2_conv1d_forward.__get__(layers.mlp.c_proj, transformers.modeling_utils.Conv1D)

        # op_fusion(layers, 'Encoder')
        hyper_sec(layers, 'Encoder')

    return model
```

SambaNova SYSTEMS

Tying it all together

- The **main()** function for generation is very similar to the previous main() for training
- The most significant difference is that for compilation, the **inference** argument will be True
- The generated predictions will be output to standard out

```python
def main(argv: List[str]) -> None:
    # Parse the args
    args = parse_app_args(argv=argv, common_parser_fn=add_common_args, run_parser_fn=add_run_args)

    # Download the model from HuggingFace
    if args.config_name:
        config = AutoConfig.from_pretrained(args.config_name, cache_dir=args.cache_dir)
        model = AutoModelForCausalLM.from_config(config)
    elif args.model_name_or_path:
        model = AutoModelForCausalLM.from_pretrained(args.model_name_or_path, cache_dir=args.cache_dir)
    else:
        raise RuntimeError("Must provide --model_name_or_path or --config_name")

    # Patch the model here
    model = patch_model(model, args)

    samba.from_torch_model_(model)

    inputs = get_compile_inputs(args)

    if args.command == 'compile':
        samba.session.compile(model, inputs, app_dir=samba.utils.get_file_dir(__file__), inference=args.inference)
    elif args.command == 'run':
        traced_outputs = utils.trace_graph(model, inputs, pef=args.pef)
        predictions = generate(args, model, traced_outputs)
        print(*predictions, sep=f"\n{'-' * 20}\n")
```

SambaNova SYSTEMS

Compile & Inference Commands

Compile Command

```
SN_NUM_THREADS=32 python tutorial_train.py compile \
--inference \
--max_seq_length 1024 \
-b 1 \
--config_name <path/to/config.json> \
--model_name_or_path gpt2 \
--num_tiles 4 \
--o0 \
--pef <name_of_pef>
```

Training Command

```
SN_NUM_THREADS=32 python tutorial_train.py run \
--inference \
--max_seq_length 1024 \
--max_tokens_to_generate 20 \
-b 1 \
--data_dir <path/to/prompts> \
--checkpoint_name <path/to/checkpoint> \
--model_name_or_path gpt2 \
--pef <path/to/pef>
```

SambaNova SYSTEMS

Some Results

REVIEW: Susie Q. is one of those rare, and sweet movies that give you a warm feeling. It's bittersweet, but wholesome, and it's characters are fun, and captivating. At first, I thought the movie would be the cliché cuddly movie that would bore me after five minutes, but was I wrong. It made me tear up at times, and it's plot was enticing, making me root for the good guys. I loved the movie, and still remember it today, 9 years later!! I recommend it highly to ANYONE, and the movie is family oriented, so you won't have to worry about unsuitable content. Truly, if Disney would show more movies that are up to par as Susie Q., it would be the most popular family oriented channel in the world. Now if only Disney would show it just ONE more time!^_^ Go Susie Q.!! QUESTION: Is this review positive or negative?

positive<|endoftext|>

REVIEW: I love his martial arts style, it is quick, close up and oh so fast, but it seems like his movies are becoming more and more crime based lifestyle quality and less meaning...I thought he was out to bring forth a deeper message. At least some of the movies had a hidden meaning or agenda this one had some good redeeming qualities of the character but the rest was so far off. I was very disappointed. The martial arts is also suffering. I am hoping to see a more devoted Segal in his future films. This movie also lacks in keeping the story line going, there are too many gaps so the thought is lost. Too many things are cryptic without a solution. QUESTION: Is this review positive or negative?

negative<|endoftext|>


SambaNova®
SYSTEMS