kernel_pruning_similarites

November 11, 2022

[ ]:
```
import torch
import load_dataset as dl
import load_model as lm
import train_model as tm
import initialize_pruning as ip
import facilitate_pruning as fp
import torch.nn.utils.prune as prune
import os  # use to access the files
from datetime import date
```

[ ]:
```
dataset_dir = '/home/pragnesh/project/Dataset/';
selected_dataset_dir = 'IntelIC'
train_folder = 'train'; test_folder = 'test'
# String Parameter for Model
loadModel = False; is_transfer_learning = False
program_name = 'vgg_net_kernel_pruning_3Aug';
model_dir = '/home/pragnesh/project/Model/'
selectedModel = 'vgg16_IntelIc_Prune'
load_path = f'{model_dir}{program_name}/{selected_dataset_dir}/{selectedModel}'
```

[ ]:
```
# String parameter to Log Output
logDir = '/home/pragnesh/project/Logs/'
folder_path = f'{logDir}{program_name}/{selected_dataset_dir}/'
logResultFile = f'{folder_path}result.log'
outFile = f'{folder_path}lastResult.log'
outLogFile = f'{folder_path}outLogFile.log'
```

[ ]:
```
if torch.cuda.is_available():
    device1 = torch.device('cuda')
else:
    device1 = torch.device('cpu')
```

[ ]:
```
def ensure_dir(dir_path):
    directory = os.path.dirname(dir_path)
    if not os.path.exists(directory):
        os.makedirs(directory)
```

[ ]:
```
ensure_dir(f'{model_dir}{program_name}/')
ensure_dir(f'{model_dir}{program_name}/{selected_dataset_dir}/')
ensure_dir(f'{logDir}{program_name}')
ensure_dir(f'{logDir}{program_name}/{selected_dataset_dir}/')
```

[ ]:
```
dl.set_image_size(224)
dl.set_batch_size = 16
dataLoaders = dl.data_loader(set_datasets_arg=dataset_dir,
                             selected_dataset_arg=selected_dataset_dir,
                             train_arg=train_folder, test_arg=test_folder)
```

[ ]:
```
if loadModel:  # Load the saved trained model
    new_model = torch.load(load_path, map_location=torch.device(device1))
else:  # Load the standard model from library
    new_model = lm.load_model(model_name='vgg16', number_of_class=6,
                              pretrainval=is_transfer_learning,
                              freeze_feature_arg=False, device_l=device1)
```

[ ]:
```
today = date.today()
d1 = today.strftime("%d-%m")
print(f"\n.........OutLog For the {d1}.............")
with open(outLogFile, 'a') as f:
    f.write(f"\n\n........OutLog For the {d1}.........\n\n")
f.close()
```

[ ]:
```
block_list = []; feature_list = []; conv_layer_index = []; module = []
prune_count = []; new_list = []; candidate_conv_layer = []
layer_number = 0; st = 0; en = 0
```

[ ]:
```
def initialize_lists_for_pruning():
    global block_list, feature_list, conv_layer_index, prune_count
    global module, new_list, candidate_conv_layer, layer_number, st, en
    block_list = ip.create_block_list(new_model)  # ip.getBlockList('vgg16')
    feature_list = ip.create_feature_list(new_model)
    conv_layer_index = ip.find_conv_index(new_model)
    module = ip.make_list_conv_param(new_model)
    prune_count = ip.get_prune_count(module=module, blocks=block_list, max_pr=.
 ↪1)
    new_list = []; layer_number = 0; st = 0; en = 0
    candidate_conv_layer = []
initialize_lists_for_pruning()
```

[ ]:
```
def compute_conv_layer_dist_kernel_pruning(module_candidate_convolution,␣
 ↪block_list_l, block_id):
    with open(outLogFile, "a") as out_file:
        out_file.write("\nExecuting Compute Candidate Convolution Layer")
    out_file.close()
```

```
    global layer_number
    candidate_convolution_layer = []
    end_index = 0
    for bl in range(len(block_list_l)):
        start_index = end_index
        end_index = end_index + block_list_l[bl]
        if bl != block_id:
            continue
        with open(outLogFile, "a") as out_file:
            out_file.write(f'\nblock ={bl} blockSize={block_list_l[bl]},␣
 ↪start={start_index}, End={end_index}')
        out_file.close()

        # newList = []
        # candidList = []
        for lno in range(start_index, end_index):
            # layer_number =st+i
            with open(outLogFile, 'a') as out_file:
                out_file.write(f"\nlno in compute candidate {lno}")
            out_file.close()
            candidate_convolution_layer.append(
                fp.compute_distance_score_kernel(
                    module_candidate_convolution[lno]._parameters['weight'],
                    n=1,dim_to_keep=[0, 1],prune_amount=prune_count[lno]))
        break
    return candidate_convolution_layer
```

[ ]:
```
'''
def compute_distance_score_kernel(tensor_t, n=1, dim_to_keep=[0, 1],␣
 ↪prune_amount=1):
    # dims = all axes, except for the one identified by `dim`
    dim_to_prune = list(range(tensor_t.dim()))  # initially it has all dims
    # remove dim which we want to keep from dimensions to prune
    for i in range(len(dim_to_keep)):
        dim_to_prune.remove(dim_to_keep[i])
    size = tensor_t.shape
    module_buffer = torch.zeros_like(tensor_t)

    # shape of norm should be equal to multiplication of dim to keep values
    norm = torch.norm(tensor_t, p=n, dim=dim_to_prune)
    size = tensor_t.shape
    for i in range(size[0]):
        for j in range(size[1]):
            module_buffer[i][j] = tensor_t[i][j] / norm[i][j]
    dist = torch.zeros(size[1], size[0], size[0])

    kernel_list_distance = []
```

```
    for j in range(size[1]):
        idx_tuple = []
        print('.', end='')
        max_value = -1
        max_idx = -1
        for i1 in range(size[0]):
            for i2 in range((i1 + 1), size[0]):
                dist[j][i1][i2] = torch.norm((module_buffer[i1][j] -␣
↪module_buffer[i2][j]), p=1)
                dist[j][i2][i1] = dist[j][i1][i2]
                if len(idx_tuple) < prune_amount:
                    idx_tuple.append([j, i1, i2, dist[j][i1][i2]])
                    idx = len(idx_tuple) - 1
                    if max_value < idx_tuple[idx][3]:
                        max_value = idx_tuple[idx][3]
                        max_idx = idx
                    continue
                if dist[j][i1][i2] < max_value:
                    del idx_tuple[max_idx]
                    idx_tuple.append([j, i1, i2, dist[j][i1][i2]])
                    max_value = idx_tuple[0][3]
                    max_idx = 0
                    for new_max_idx in range(1, len(idx_tuple)):
                        if max_value < idx_tuple[new_max_idx][3]:
                            max_value = idx_tuple[new_max_idx][3]
                            max_idx = new_max_idx
        kernel_list_distance.append(idx_tuple)
    return kernel_list_distance '''
```

[ ]:
```
class KernelPruningSimilarities(prune.BasePruningMethod):
    PRUNING_TYPE = 'unstructured'
    def compute_mask(self, t, default_mask):
        with open(outLogFile, "a") as log_file:
            log_file.write("\n Executing Compute Mask")
        log_file.close()
        mask = default_mask.clone()
        # mask.view(-1)[::2] = 0
        size = t.shape
        print(f"\n{size}")
        with open(outLogFile, "a") as log_file:
            log_file.write(f'\nLayer Number:{layer_number} \nstart={st}␣
↪\nlength of new list={len(new_list)}')
        log_file.close()
        for k1 in range(len(new_list)):
            for k2 in range(len(new_list[layer_number - st][k1])):
                i = new_list[layer_number - st][k1][k2][1]
                j = new_list[layer_number - st][k1][k2][0]
```

```
                if k1 == j:
                    print(":", end='')
                mask[i][j] = 0
        return mask
```

[ ]:
```
def kernel_unstructured_similarities(kernel_module, name):
    ChannelPruningMethodSimilarities.apply(kernel_module, name)
    return kernel_module
```

[ ]:
```
def iterative_kernel_pruning_dist_block_wise(new_model_arg, prune_module,
                                             block_list_l, prune_epochs):
    with open(outLogFile, "a") as out_file:
        out_file.write("\nPruning Process Start")
    out_file.close()
    # pc = [1, 3, 9, 26, 51]
    global new_list
    for e in range(prune_epochs):
        start = 0
        end = len(block_list_l)
        for blkId in range(start, end):
            # 2 Compute distance between kernel for candidate conv layer
            new_list = compute_conv_layer_dist_kernel_pruning(
                module_candidate_convolution=prune_module,
                block_list_l=block_list_l, block_id=blkId)
            # 5 perform Custom pruning where we mask the prune weight
            for j in range(block_list_l[blkId]):
                if blkId < 2:
                    layer_number_to_prune = (blkId * 2) + j
                else:  # blkId >= 2:
                    layer_number_to_prune = 4 + (blkId - 2) * 3 + j
                kernel_unstructured_similarities(
                    kernel_module=prune_module[layer_number_to_prune],
                    name='weight')
            new_list = None
        # 6.  Commit Pruning
        with open(outLogFile, 'a') as out_file:
            out_file.write("\ncommit the pruning")
        out_file.close()
        for i in range(len(prune_module)):
            prune.remove(module=prune_module[i], name='weight')
        # 7.  Update feature list
        global feature_list
        feature_list = update_feature_list(
            feature_list, prune_count, start=0, end=len(prune_count))
        # 8.  Create new temp model with updated feature list
        temp_model = lm.create_vgg_from_feature_list(
            vgg_feature_list=feature_list, batch_norm=True)
```

```
        temp_model.to(device1)
        # 9.  Perform deep copy
        lm.freeze(temp_model, 'vgg16')
        deep_copy(temp_model, new_model_arg)
        lm.unfreeze(temp_model)
        # 10.  Train pruned model
        with open(outLogFile, 'a') as out_file:
            out_file.write('\n ...Deep Copy Completed...')
            out_file.write('\n Fine tuning started....')
        out_file.close()

        tm.fit_one_cycle( dataloaders=dataLoaders,
            train_dir=dl.train_directory, test_dir=dl.test_directory,
            # Select a variant of VGGNet
            model_name='vgg16', model=temp_model, device_l=device1,
            # Set all the Hyper-Parameter for training
            epochs=8, max_lr=0.001, weight_decay=0.01, L1=0.01, grad_clip=0.1,
            opt_func=opt_func,
            log_file=logResultFile)
        with open(outLogFile, 'a') as out_file:
            out_file.write('....Fine tuning completed\n')
        out_file.close()

        save_path = f'{model_dir}{program_name}/{selected_dataset_dir}/
↪vgg16_IntelIc_Prune_{e}_b_train'
        torch.save(temp_model, save_path)
        # # # 10. Evaluate the pruned model
        train_accuracy = 0.0
        test_accuracy = 0.0

        with open(outFile, 'a') as out_file:
            out_file.write(f'\n output of the {e}th iteration is written␣
↪below\n')
            out_file.write(f'\n Train Accuracy: {train_accuracy}'
                           f'\n Test Accuracy  :  {test_accuracy} \n')
        out_file.close()

        save_path = f'{model_dir}{program_name}/selected/dataset_dir/
↪vgg16_IntelIc_Prune_{e}_b_train'
        # save_path = f'/home3/pragnesh/Model/vgg16_IntelIc_Prune_{e}_b_train'
        torch.save(temp_model, save_path)
```

[ ]:
```
initialize_lists_for_pruning()
iterative_kernel_pruning_dist_block_wise(new_model_arg=new_model,
    prune_module=module, block_list_l=block_list, prune_epochs=6)
```