#_ important PyTorch Operations [+100]

Basics and Tensor Operations:

- `torch.tensor(data)`: Create a PyTorch tensor from data.
- `torch.zeros(size)`: Create a tensor filled with zeros.
- `torch.ones(size)`: Create a tensor filled with ones.
- `torch.rand(size)`: Create a tensor with random values uniformly distributed between 0 and 1.
- `torch.randn(size)`: Create a tensor with random values from a normal distribution.
- `torch.arange(start, end, step)`: Create a 1-D tensor of size (end - start) / step.
- `tensor.size()`: Get the size of the tensor.
- `tensor.view(size)`: Resize the tensor.
- `tensor.reshape(size)`: Reshape the tensor.
- `tensor.numpy()`: Convert tensor to a NumPy array.
- `torch.from_numpy(ndarray)`: Create a tensor from a NumPy array.
- `tensor.to(device)`: Move tensor to a specified device (CPU or GPU).
- `tensor.type(dtype)`: Cast the tensor to a specified type.

Math **Operations:**

- `torch.add(x, y)`: Element-wise addition of tensors.
- `torch.sub(x, y)`: Element-wise subtraction of tensors.
- `torch.mul(x, y)`: Element-wise multiplication of tensors.
- `torch.div(x, y)`: Element-wise division of tensors.
- `torch.matmul(x, y)`: Matrix multiplication.
- `torch.mm(x, y)`: Alias for torch.matmul for 2D tensors.
- `torch.dot(x, y)`: Dot product of two tensors.
- `torch.exp(tensor)`: Exponential of each element in the tensor.
- `torch.log(tensor)`: Natural logarithm of each element in the tensor.
- `torch.pow(tensor, exponent)`: Power of each element in the tensor.
- `torch.sqrt(tensor)`: Square root of each element in the tensor.
- `torch.abs(tensor)`: Absolute value of each element in the tensor.

By: Waleed Mousa

- `torch.sum(tensor)`: Sum of all elements in the tensor.
- `torch.mean(tensor)`: Mean of all elements in the tensor.
- `torch.max(tensor)`: Maximum value in the tensor.
- `torch.min(tensor)`: Minimum value in the tensor.
- `torch.std(tensor)`: Standard deviation of the tensor.

Indexing, Slicing, Joining:

- `tensor[index]`: Access elements using Python indexing.
- `tensor[start: end: step]`: Slice the tensor.
- `torch.cat(tensors, dim)`: Concatenate tensors along a dimension.
- `torch.stack(tensors, dim)`: Stack tensors along a new dimension.
- `torch.chunk(tensor, chunks, dim)`: Split tensor into a specific number of chunks.
- `torch.split(tensor, split_size, dim)`: Split tensor into sections.

Gradient and Computational Graph:

- `tensor.requires_grad_()`: Enable tracking of gradient for the tensor.
- `tensor.backward()`: Compute the gradient of the tensor.
- `tensor.grad`: Access the gradient of the tensor.
- `with torch.no_grad()`: Context manager to disable gradient computation.

Neural Network Building Blocks:

- `torch.nn.Linear(in_features, out_features)`: Fully connected layer.
- `torch.nn.Conv2d(in_channels, out_channels, kernel_size)`: 2D convolution layer.
- `torch.nn.MaxPool2d(kernel_size, stride)`: 2D max pooling layer.
- `torch.nn.ReLU()`: ReLU activation function.
- `torch.nn.Sigmoid()`: Sigmoid activation function.
- `torch.nn.Tanh()`: Tanh activation function.
- `torch.nn.BatchNorm2d(num_features)`: Batch normalization for 2D inputs.
- `torch.nn.Dropout(p)`: Dropout layer.

By: Waleed Mousa

Loss Functions:

- `torch.nn.MSELoss()`: Mean squared error loss.
- `torch.nn.CrossEntropyLoss()`: Cross-entropy loss for multi-class classification.
- `torch.nn.BCELoss()`: Binary cross-entropy loss.
- `torch.nn.BCEWithLogitsLoss()`: Binary cross-entropy loss with logits.
- `torch.nn.NLLLoss()`: Negative log-likelihood loss.

Optimizers:

- `torch.optim.SGD(params, lr)`: Stochastic Gradient Descent optimizer.
- `torch.optim.Adam(params, lr)`: Adam optimizer.
- `torch.optim.Adagrad(params, lr)`: Adagrad optimizer.
- `torch.optim.RMSprop(params, lr)`: RMSprop optimizer.

Data Handling:

- `torch.utils.data.Dataset`: Abstract class representing a dataset.
- `torch.utils.data.DataLoader(dataset)`: Data loader for a dataset.
- `torchvision.transforms`: Common image transformations for pre-processing.
- `torchvision.datasets`: Datasets for common vision tasks.

Model Training and Evaluation:

- `model.train()`: Set the model to training mode.
- `model.eval()`: Set the model to evaluation mode.
- `torch.save(model.state_dict(), path)`: Save model state.
- `model.load_state_dict(torch.load(path))`: Load model state.

GPU Operations:

- `torch.cuda.is_available()`: Check if CUDA is available.
- `tensor.cuda()`: Move tensor to GPU.
- `model.cuda()`: Move model to GPU.

By: Waleed Mousa

- `torch.device("cuda:0")`: Specify which GPU to use.

Autograd and Function:

- `torch.autograd.Variable(tensor)`: Deprecated, use tensors with `requires_grad`.
- `torch.autograd.grad(outputs, inputs)`: Compute gradients w.r.t inputs.
- `torch.autograd.Function`: Base class for custom autograd Functions.

Advanced Tensor Operations:

- `torch.unbind(tensor, dim)`: Removes a tensor dimension.
- `torch.squeeze(tensor)`: Remove all dimensions of size 1.
- `torch.unsqueeze(tensor, dim)`: Add a dimension of size 1.
- `torch.transpose(tensor, dim0, dim1)`: Transpose two dimensions of a tensor.
- `torch.t(tensor)`: Transpose for 2D tensors.
- `torch.einsum(equation, tensors)`: Perform Einstein summation.
- `torch.norm(tensor, p)`: Compute the p-norm.
- `torch.clamp(tensor, min, max)`: Clamp tensor values to a range.

Randomness and Probability:

- `torch.manual_seed(seed)`: Manually set the random seed for reproducibility.
- `torch.randperm(n)`: Random permutation of integers from 0 to n-1.
- `torch.bernoulli(tensor)`: Draw binary random numbers from a Bernoulli distribution.

Tensor Serialization:

- `torch.save(tensor, path)`: Save tensor to file.
- `torch.load(path)`: Load tensor from file.

Functional Interface:

- `torch.nn.functional.relu(tensor)`: Functional interface for ReLU.

By: Waleed Mousa

- `torch.nn.functional.conv2d(input, weight)`: Functional interface for 2D convolution.
- `torch.nn.functional.max_pool2d(input, kernel_size)`: Functional interface for 2D max pooling.
- `torch.nn.functional.cross_entropy(input, target)`: Functional interface for cross-entropy loss.

Custom Modules and Models:

- `class MyModule(torch.nn.Module)`: Define a custom neural network module.
- `def forward(self, x)`: Define the forward pass of a module.

Tensor Decompositions and Linear Algebra:

- `torch.svd(tensor)`: Singular value decomposition.
- `torch.eig(tensor)`: Compute the eigenvalues and eigenvectors.
- `torch.inverse(tensor)`: Compute the inverse of a matrix.

Parallel and Distributed Computing:

- `torch.nn.DataParallel(model)`: Data parallelism over multiple GPUs.
- `torch.distributed.init_process_group(backend)`: Initialize the distributed backend for multi-process parallelism.

Tensor Inspection and Debugging:

- `tensor.type()`: Get the data type of the tensor.
- `tensor.device`: Get the device of the tensor.
- `tensor.layout`: Get the memory layout of the tensor.
- `tensor.is_cuda`: Check if the tensor is on CUDA.
- `tensor.requires_grad`: Check if the tensor requires gradient.
- `torch.set_printoptions(precision)`: Set printing options for tensors.

Advanced GPU and CUDA Operations:

- `torch.cuda.memory_allocated()`: Get the current GPU memory usage.

By: Waleed Mousa

- `torch.cuda.memory_cached()`: Get the cached GPU memory.
- `torch.cuda.empty_cache()`: Release cached memory.
- `torch.cuda.get_device_name(device)`: Get the name of the GPU device.

Quantization for Model Optimization:

- `torch.quantization.quantize_dynamic(model, dtype)`: Apply dynamic quantization to a model.
- `torch.quantization.quantize_static(model, calibration, dtype)`: Apply static quantization to a model.
- `torch.quantization.QuantStub()`: Quantization stub for quantizable modules.
- `torch.quantization.DeQuantStub()`: Dequantization stub for quantizable modules.
- `torch.quantization.prepare(model)`: Prepare the model for quantization calibration.
- `torch.quantization.convert(model)`: Convert the model to a quantized version.

Tensor Advanced Operations:

- `torch.gather(input, dim, index)`: Gather values along an axis specified by dim.
- `torch.scatter(input, dim, index, src)`: Write values from `src` into `input` at positions specified by `index`.
- `torch.repeat_interleave(tensor, repeats, dim)`: Repeat elements of a tensor along a dimension.
- `torch.roll(tensor, shifts, dims)`: Roll the elements of a tensor along a given dimension.

Functional API for Complex Operations:

- `torch.nn.functional.dropout(input, p, training)`: Apply dropout to the input.
- `torch.nn.functional.interpolate(input, size, mode)`: Upsample or downsample a tensor.
- `torch.nn.functional.pad(input, pad, mode, value)`: Pad a tensor.

By: Waleed Mousa

- `torch.nn.functional.normalize(input, p, dim)`: Normalize a tensor.

Debugging and Profiling:

- `torch.autograd.set_detect_anomaly(True)`: Enable anomaly detection for debugging.
- `torch.autograd.profiler.profile()`: Context manager for profiling the performance of PyTorch operations.

PyTorch Extensions and Utilities:

- `torch.utils.checkpoint.checkpoint(function, *args)`: Enable checkpointing for memory-efficient gradients.
- `torch.utils.data.random_split(dataset, lengths)`: Randomly split a dataset into non-overlapping new datasets of given lengths.
- `torch.utils.tensorboard.SummaryWriter(log_dir)`: Log PyTorch models and metrics into TensorBoard.

Advanced Model Building:

- `torch.nn.utils.rnn.pack_padded_sequence(input, lengths, batch_first)`: Pack a tensor containing padded sequences for RNN processing.
- `torch.nn.utils.rnn.pad_packed_sequence(sequence, batch_first)`: Pad packed batch of variable length sequences.
- `torch.nn.utils.prune.l1_unstructured(module, name, amount)`: Apply L1 unstructured pruning to a module.

Custom Autograd Functions:

- `class MyFunction(torch.autograd.Function)`: Define a custom autograd function.
- `@staticmethod def forward(ctx, input)`: Forward pass for custom autograd function.
- `@staticmethod def backward(ctx, grad_output)`: Backward pass for custom autograd function.

Interoperability with NumPy:

By: Waleed Mousa

- `torch.from_numpy(numpy_array)`: Create a tensor from a NumPy array.
- `tensor.detach().numpy()`: Convert a tensor to a NumPy array.

Distributed Training:

- `torch.distributed.init_process_group(backend, world_size, rank)`: Initialize distributed process group for parallel training.
- `torch.nn.parallel.DistributedDataParallel(model)`: Wrap model for distributed training.

TensorBoard Integration:

- `from torch.utils.tensorboard import SummaryWriter`: Import TensorBoard SummaryWriter for logging.
- `writer.add_scalar('tag', value, step)`: Log a scalar variable.
- `writer.add_histogram('tag', values, step)`: Log values as a histogram.
- `writer.add_image('tag', img_tensor, step)`: Log an image.
- `writer.add_graph(model, input_to_model)`: Log model graph.

Advanced CUDA **Operations**:

- `torch.cuda.stream(stream)`: Context manager to select a stream.
- `torch.cuda.amp.autocast()`: Automatic mixed precision context manager.
- `torch.cuda.amp.GradScaler()`: Gradient scaler for mixed precision training.

Custom Datasets **and** Data Loaders:

- `class CustomDataset(torch.utils.data.Dataset)`: Define a custom dataset.
- `def __len__(self)`: Return the size of the dataset.
- `def __getitem__(self, idx)`: Retrieve an item by index.

Serialization and Saving Models:

- `torch.save(object, path)`: Serialize a PyTorch object to disk.

By: Waleed Mousa

- `torch.load(path)`: Deserialize a PyTorch object from disk.
- `model.state_dict()`: Get model's state dictionary.
- `model.load_state_dict(state_dict)`: Load a state dictionary into the model.

Working with Hooks:

- `model.register_forward_hook(hook)`: Register a forward hook on the model.
- `model.register_backward_hook(hook)`: Register a backward hook on the model.

Model Fine-Tuning and Transfer Learning:

- `torchvision.models.resnet18(pretrained=True)`: Load a pre-trained ResNet-18 model.
- `for param in model.parameters(): param.requires_grad = False`: Freeze parameters for transfer learning.

PyTorch for Mobile:

- `torch.jit.script(model)`: Convert model to TorchScript for mobile deployment.
- `torch.jit.save(scripted_model, path)`: Save a scripted model for mobile use.

Dynamic Computational Graphs:

- `torch.autograd.Variable(tensor, requires_grad)`: Wrapper around tensors for autograd.
- `variable.grad_fn`: Access grad function of the variable.

Specialized Layers and Functions:

- `torch.nn.Embedding(num_embeddings, embedding_dim)`: A simple lookup table that stores embeddings of a fixed dictionary and size.
- `torch.nn.MultiheadAttention(embed_dim, num_heads)`: Multi-head attention mechanism.

By: Waleed Mousa

- `torch.nn.utils.weight_norm(module, name)`: Apply weight normalization to a module.

Memory Management and Optimization:

- `torch.cuda.memory_summary(device=None, abbreviated=False)`: Print a summary of CUDA memory allocation and usage.

Dynamic Quantization:

- `torch.quantization.quantize_dynamic(model, {torch.nn.LSTM, torch.nn.Linear}, dtype=torch.qint8)`: Dynamically quantize LSTM and Linear layers for efficiency.