[Deep Learning Using PyTorch] [cheatsheet]

Tensor Creation and Manipulation

- Create a tensor from a list: `tensor = torch.tensor([1, 2, 3])`
- Create a tensor of zeros: `tensor = torch.zeros(shape)`
- Create a tensor of ones: `tensor = torch.ones(shape)`
- Create a tensor with random values: `tensor = torch.rand(shape)`
- Create a tensor with normally distributed random values: `tensor = torch.randn(shape)`
- Create a tensor with a range of values: `tensor = torch.arange(start, end, step)`
- Create a tensor with evenly spaced values: `tensor = torch.linspace(start, end, steps)`
- Reshape a tensor: `tensor = tensor.view(new_shape)`
- Transpose a tensor: `tensor = tensor.transpose(dim1, dim2)`
- Flatten a tensor: `tensor = tensor.flatten()`
- Concatenate tensors along a dimension: `tensor = torch.cat([tensor1, tensor2], dim)`
- Stack tensors along a new dimension: `tensor = torch.stack([tensor1, tensor2], dim)`
- Squeeze a tensor (remove dimensions of size 1): `tensor = tensor.squeeze()`
- Unsqueeze a tensor (add a dimension of size 1): `tensor = tensor.unsqueeze(dim)`
- Permute the dimensions of a tensor: `tensor = tensor.permute(dims)`

Tensor Operations

- Addition: `result = tensor1 + tensor2`
- Subtraction: `result = tensor1 - tensor2`
- Multiplication (element-wise): `result = tensor1 * tensor2`
- Division (element-wise): `result = tensor1 / tensor2`
- Matrix multiplication: `result = tensor1.matmul(tensor2)`
- Exponential: `result = torch.exp(tensor)`
- Logarithm: `result = torch.log(tensor)`
- Square root: `result = torch.sqrt(tensor)`
- Sine: `result = torch.sin(tensor)`
- Cosine: `result = torch.cos(tensor)`

By: Waleed Mousa

- Tangent: `result = torch.tan(tensor)`
- Sigmoid: `result = torch.sigmoid(tensor)`
- ReLU: `result = torch.relu(tensor)`
- Tanh: `result = torch.tanh(tensor)`
- Softmax: `result = torch.softmax(tensor, dim)`

Neural Network Layers

- Linear layer: `layer = nn.Linear(in_features, out_features)`
- Convolutional layer: `layer = nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)`
- Transposed convolutional layer: `layer = nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride, padding)`
- Max pooling layer: `layer = nn.MaxPool2d(kernel_size, stride, padding)`
- Average pooling layer: `layer = nn.AvgPool2d(kernel_size, stride, padding)`
- Batch normalization layer: `layer = nn.BatchNorm2d(num_features)`
- Dropout layer: `layer = nn.Dropout(p)`
- Recurrent layer (RNN): `layer = nn.RNN(input_size, hidden_size, num_layers)`
- Long Short-Term Memory layer (LSTM): `layer = nn.LSTM(input_size, hidden_size, num_layers)`
- Gated Recurrent Unit layer (GRU): `layer = nn.GRU(input_size, hidden_size, num_layers)`
- Embedding layer: `layer = nn.Embedding(num_embeddings, embedding_dim)`

Loss Functions

- Mean Squared Error (MSE) loss: `loss_fn = nn.MSELoss()`
- Cross-Entropy loss: `loss_fn = nn.CrossEntropyLoss()`
- Binary Cross-Entropy loss: `loss_fn = nn.BCELoss()`
- Negative Log-Likelihood loss: `loss_fn = nn.NLLLoss()`
- Kullback-Leibler Divergence loss: `loss_fn = nn.KLDivLoss()`
- Margin Ranking loss: `loss_fn = nn.MarginRankingLoss()`
- Triplet Margin loss: `loss_fn = nn.TripletMarginLoss()`
- Cosine Embedding loss: `loss_fn = nn.CosineEmbeddingLoss()`
- Hinge Embedding loss: `loss_fn = nn.HingeEmbeddingLoss()`

Optimization Algorithms

- Stochastic Gradient Descent (SGD): `optimizer = torch.optim.SGD(model.parameters(), lr)`
- Adam: `optimizer = torch.optim.Adam(model.parameters(), lr)`
- RMSprop: `optimizer = torch.optim.RMSprop(model.parameters(), lr)`
- Adagrad: `optimizer = torch.optim.Adagrad(model.parameters(), lr)`
- Adadelta: `optimizer = torch.optim.Adadelta(model.parameters(), lr)`
- Adamax: `optimizer = torch.optim.Adamax(model.parameters(), lr)`
- Sparse Adam: `optimizer = torch.optim.SparseAdam(model.parameters(), lr)`
- LBFGS: `optimizer = torch.optim.LBFGS(model.parameters(), lr)`

Learning Rate Schedulers

- Step LR: `scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size, gamma)`
- Multi-Step LR: `scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones, gamma)`
- Exponential LR: `scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma)`
- Cosine Annealing LR: `scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max)`
- Reduce LR on Plateau: `scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode, factor, patience)`
- Cyclic LR: `scheduler = torch.optim.lr_scheduler.CyclicLR(optimizer, base_lr, max_lr, step_size_up)`

Model Training and Evaluation

- Move model to device: `model = model.to(device)`
- Set model to training mode: `model.train()`
- Set model to evaluation mode: `model.eval()`
- Forward pass: `outputs = model(inputs)`
- Compute loss: `loss = loss_fn(outputs, targets)`
- Backward pass: `loss.backward()`
- Update model parameters: `optimizer.step()`
- Zero gradients: `optimizer.zero_grad()`
- Get model parameters: `parameters = model.parameters()`
- Get model state dictionary: `state_dict = model.state_dict()`
- Load model state dictionary: `model.load_state_dict(state_dict)`
- Save model checkpoint: `torch.save(model.state_dict(), 'checkpoint.pth')`

- Load model checkpoint: `model.load_state_dict(torch.load('checkpoint.pth'))`

Data Loading and Processing

- Create a dataset: `dataset = torch.utils.data.TensorDataset(inputs, targets)`
- Create a data loader: `data_loader = torch.utils.data.DataLoader(dataset, batch_size, shuffle)`
- Iterate over data loader: `for batch in data_loader: inputs, targets = batch`
- Normalize data: `data = (data - data.mean()) / data.std()`
- Resize images: `images = torch.nn.functional.interpolate(images, size)`
- Random crop images: `images = torchvision.transforms.RandomCrop(size)(images)`
- Random horizontal flip images: `images = torchvision.transforms.RandomHorizontalFlip()(images)`
- Convert images to tensors: `images = torchvision.transforms.ToTensor()(images)`
- Normalize images: `images = torchvision.transforms.Normalize(mean, std)(images)`

Pretrained Models

- Load a pretrained model: `model = torchvision.models.resnet18(pretrained=True)`
- Freeze model weights: `for param in model.parameters(): param.requires_grad = False`
- Replace the last layer of a pretrained model: `model.fc = nn.Linear(512, num_classes)`
- Extract features from a pretrained model: `features = model(inputs)`

Model Evaluation Metrics

- Accuracy: `accuracy = (predicted == targets).float().mean()`
- Precision: `precision = torch.sum(predicted * targets) / torch.sum(predicted)`
- Recall: `recall = torch.sum(predicted * targets) / torch.sum(targets)`
- F1 score: `f1_score = 2 * (precision * recall) / (precision + recall)`
- Mean Absolute Error (MAE): `mae = torch.abs(predicted - targets).mean()`

By: Waleed Mousa

- Mean Squared Error (MSE): `mse = torch.square(predicted - targets).mean()`
- Root Mean Squared Error (RMSE): `rmse = torch.sqrt(mse)`
- Intersection over Union (IoU): `iou = torch.sum(predicted * targets) / torch.sum((predicted + targets) > 0)`
- Area Under the ROC Curve (AUC): `auc = torchmetrics.functional.auroc(predicted, targets)`
- Average Precision (AP): `ap = torchmetrics.functional.average_precision(predicted, targets)`
- Confusion Matrix: `cm = torchmetrics.functional.confusion_matrix(predicted, targets)`

Model Visualization

- Visualize the model architecture: `print(model)`
- Visualize the model graph: `torchviz.make_dot(outputs, params=dict(model.named_parameters())).render('model_graph', format='png')`
- Visualize the model summary: `torchsummary.summary(model, input_size)`

Transfer Learning

- Freeze the weights of the feature extractor: `for param in model.features.parameters(): param.requires_grad = False`
- Fine-tune the last layer: `for param in model.classifier.parameters(): param.requires_grad = True`
- Load a pretrained model and replace the last layer: `model = torchvision.models.resnet18(pretrained=True); model.fc = nn.Linear(512, num_classes)`

Adversarial Attacks

- Fast Gradient Sign Method (FGSM) attack: `perturbed_inputs = inputs + epsilon * torch.sign(inputs.grad)`
- Projected Gradient Descent (PGD) attack: `for _ in range(num_steps): perturbed_inputs = torch.clamp(perturbed_inputs + alpha * torch.sign(perturbed_inputs.grad), min=inputs-epsilon, max=inputs+epsilon)`
- Carlini & Wagner (C&W) attack: `adversarial_inputs = torch.clamp(inputs + perturbations, min=0, max=1)`

By: Waleed Mousa

Model Pruning

- Prune model weights: `pruned_model = torch.nn.utils.prune.random_unstructured(model, name='weight', amount=pruning_ratio)`
- Prune model biases: `pruned_model = torch.nn.utils.prune.l1_unstructured(model, name='bias', amount=pruning_ratio)`
- Prune model layers: `pruned_model = torch.nn.utils.prune.ln_structured(model, name='conv', amount=pruning_ratio, n=2, dim=0)`

Model Quantization

- Quantize model weights: `quantized_model = torch.quantization.quantize_dynamic(model, {torch.nn.Linear}, dtype=torch.qint8)`
- Quantize model activations: `quantized_model = torch.quantization.quantize_dynamic(model, {torch.nn.ReLU}, dtype=torch.quint8)`
- Convert model to quantized version: `quantized_model = torch.quantization.convert(model)`

Distributed Training

- Initialize distributed training: `torch.distributed.init_process_group(backend='nccl', init_method='tcp://localhost:23456', rank=args.rank, world_size=args.world_size)`
- Wrap model with DistributedDataParallel: `model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[args.local_rank])`
- Synchronize gradients across devices: `torch.distributed.barrier()`
- Reduce gradients across devices: `torch.distributed.all_reduce(tensor, op=torch.distributed.ReduceOp.SUM)`

Model Interpretability

- Compute gradients w.r.t. inputs: `gradients = torch.autograd.grad(outputs, inputs, grad_outputs=torch.ones_like(outputs))`

- Compute saliency maps: `saliency_maps = torch.abs(gradients).max(dim=1, keepdim=True)[0]`
- Compute guided backpropagation: `guided_backprop = torch.clamp(gradients, min=0)`
- Compute class activation maps (CAM): `cam = torch.sum(features * weights.view(num_classes, -1), dim=1).view(batch_size, -1)`
- Compute Grad-CAM: `grad_cam = torch.sum(features * gradients.view(batch_size, num_channels, -1), dim=2).view(batch_size, num_channels, 1, 1)`

Model Debugging

- Print model gradients: `for name, param in model.named_parameters(): if param.requires_grad: print(name, param.grad)`
- Print model activations: `for name, module in model.named_modules(): if isinstance(module, nn.ReLU): module.register_forward_hook(lambda module, input, output: print(name, output))`
- Print model parameters: `for name, param in model.named_parameters(): print(name, param)`
- Print model buffers: `for name, buffer in model.named_buffers(): print(name, buffer)`
- Set anomaly detection for debugging: `torch.autograd.set_detect_anomaly(True)`