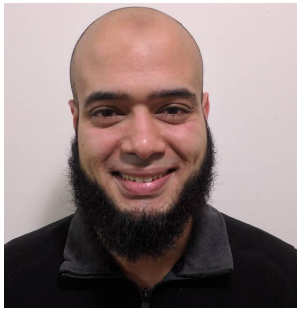
Machine Learning

PyTorch Simple Network

Mostafa S. Ibrahim
Teaching, Training and Coaching for more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / MSc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



Simple Regressor

- We usually have 2 parts
- Function to create the layers
  - We have 3 hidden layers
  - We use relu
- A feedforward function
  - Just apply sequentially
  - Typically no activation after the last layer for regression

```
class SimpleNet(nn.Module):
    def __init__(self, input_dim):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(input_dim, 30)
        self.fc2 = nn.Linear(30, 15)
        self.fc3 = nn.Linear(15, 1)
        self.activate = torch.relu

    def forward(self, x):
        x = self.fc1(x)
        x = self.activate(x)
        x = self.fc2(x)
        x = self.activate(x)
        x = self.fc3(x)
        return x
```

nn.Sequential

- The nn.Sequential is a container module that stacks layers and run in a modular simple way
  - We typically **group relevant layers together** with it
  - Whenever you update its layers, **you don't update the feedforward**!
- For complex networks, we better create a custom class that inherits from nn.Module,

```
class SimpleNet(nn.Module):
    def __init__(self, input_dim):
        super(SimpleNet, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, 30),
            nn.ReLU(),
            nn.Linear(30, 15),
            nn.ReLU(),
            nn.Linear(15, 1)
        )

    def forward(self, x):
        return self.model(x)
```

Optimizer

- model.parameters() method returns an **iterator** of all the **trainable parameters** (weights and biases) of a model.
- We pass these parameters to an optimizer object. There are many optimizers.
- AdamW is one of the greatest optimizers that work well on many problems
  - So always start with it
- You may also try SGD (what we learned), but be careful on the LR

```
input_dim = 65
model = SimpleNet(input_dim)

optimizer = optim.AdamW(model.parameters(), lr=0.001)
# try lr=0.01
#optimizer = optim.SGD(model.parameters(), lr=0.0001)
```

Loss Functions

- PyTorch has many loss functions
- Mean Squared Error Loss (MSE): **criterion = nn.MSELoss()**
- Smooth L1 Loss: criterion = nn.SmoothL1Loss()
  - A blend of L1 and L2 loss functions. Useful for regression tasks where outliers are present.
- Cross-Entropy Loss: **criterion = nn.CrossEntropyLoss()**
  - DON'T APPLY softmax. Pass the **logits** directly
  - Combines nn.LogSoftmax() and nn.NLLLoss() in one single class.
- Binary Cross-Entropy Loss: criterion = nn.BCELoss()
- BCE with Logits Loss: **criterion = nn.BCEWithLogitsLoss()**
  - DON'T APPLY Sigmoid. Pass the **logits** directly
  - Combines a Sigmoid layer and the BCE loss in one single class: numerically more stable.

Loss Combinations

- When you apply e.g. MSE on (x, y) where each is tensor of 10x20
- What happens, the get the average of the MSE
  - For each value, compute (a-b)^2
  - Sum all
  - Divide by size
- If you have multiple losses,
  - You better compute their (weighted) average to get a sense of the average

Data

- Let’s generate a trivial dataset

```
x_train = torch.rand(1000, input_dim)   # 1000 examples
y_train = 5 * torch.sum(x_train, dim=1)

criterion = nn.MSELoss()
```

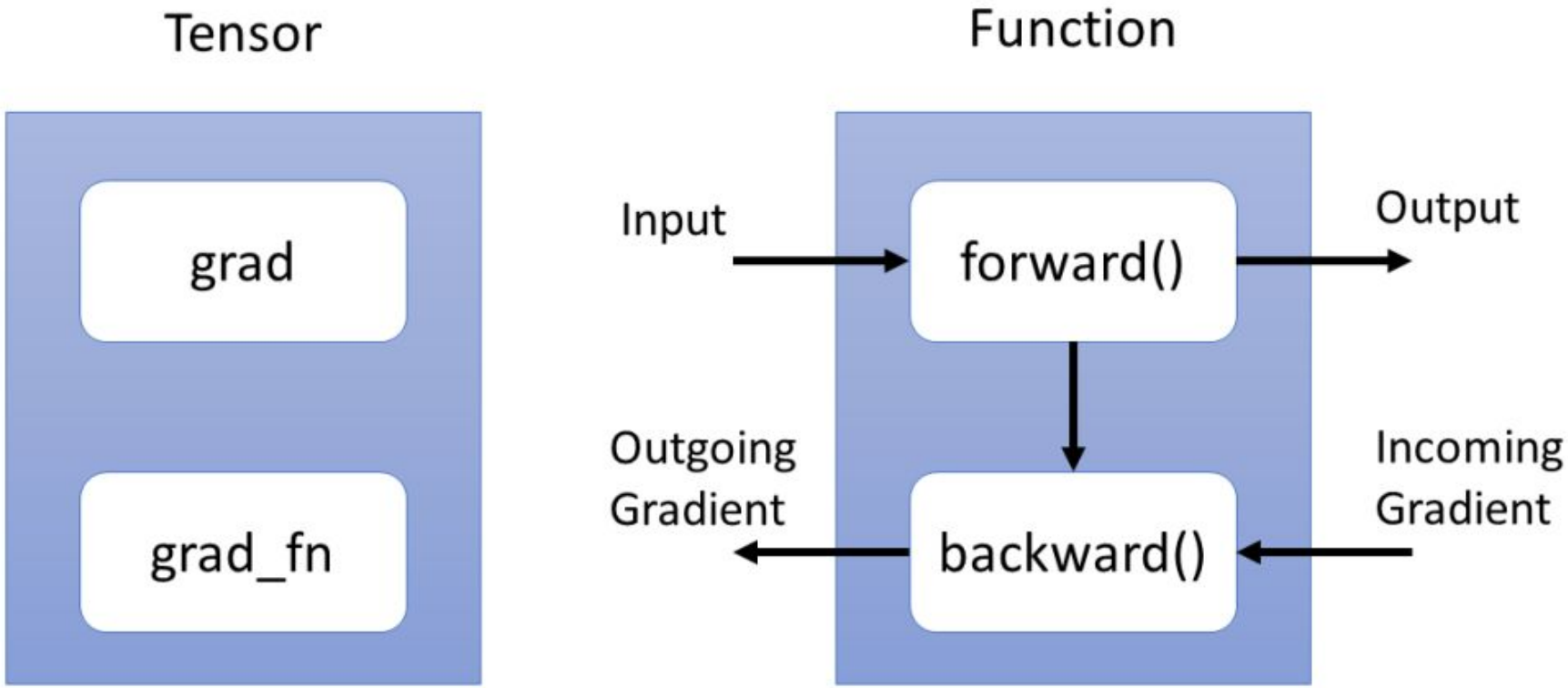
Training

- The training procedure is typically copy-paste
  - We will see another way for epochs

```
# Training loop
for epoch in range(500):
    # Forward pass
    outputs = model(x_train)
    loss = criterion(outputs, y_train)

    # Zero gradients, backward pass, optimizer step
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    print(f"Epoch {epoch+1}, Batch Loss: {loss.item()}")
    # observe this loss sums all the batch.
    # you can average for a single example average loss
```


Tensor
grad
grad_fn
Function
Input
forward()
Output
Outgoing
Gradient
backward()
Incoming
Gradient

Img src

Evaluation

- Observe below .eval() and .no_grad()
- Tip: Start the inference on samples of size one to not be confused by batch calculations
  - Then move to batch-based inference

```
# Evaluation
model.eval()  # Set the model to evaluation mode
with torch.no_grad():
    for i in range(10):
        x_test = torch.rand(1, input_dim)  # 10 test samples
        y_test = model(x_test)
        print(f"Prediction: {y_test} vs gt {5 * torch.sum(x_test, dim=1)}", )
```

Model Evaluation Settings

- Whenever we are in evaluation phase, we do 2 operations together
- model.eval():
  - There are some layers that behaves in training different than inference
    - And it performs worse if you used them in reference with training status
    - 2 popular layers: batchnorm or dropout layers
  - This layers notifies the model to use them in inference mode
  - If needed, you can go back to training mode using model.train()
- torch.no_grad(): the autograd engine won't compute the gradients
  - This will reduce memory usage and speed up computations
- There could be narrow inference cases where you want gradients computed but don't affect the weights (then only use model.eval())

Question

- If you have 100 examples to do inference on them?
- Will you batch as much as you can in a single run? Or evaluate them one by one?

- It is much more efficient to evaluate in bigger batches, however they require more memory. So choose your **maximum for inference**!

Question

- If you are **training** a network and can choose batches of sizes {1, 128, 256, 1024, 2048, 4096}. Which one you choose and why?

- In practice, you may find very big batches harm the performance!
- Start with medium size batches like 128/256: good balance between memory and speed + Stable Convergence
- There are some research tricks on how to use bigger batches

Saving a model

- There are 2 main ways
- 1) Save **entire** model (not recommended)
  - save the entire module using Python's **pickle** utility.
- 2) Save model **weights** (recommended):
  - save only the model's parameters in a dictionary

```
# Save entire model - limited loading
torch.save(model, 'model.pth')

# Save model weights (recommended): save only the model's parameters
torch.save(model.state_dict(), 'model_weights.pth')
```

Loading a model

- You can just load the model according to the way you saved it

```
model1 = torch.load('model.pth')
print(model1)

input_dim = 65
model2 = SimpleNet(input_dim)   # reconstruct
model2.load_state_dict(torch.load('model_weights.pth'))

print(model2)
```

Why not just saving the entire model?

- Whatever pickle issues
  - Known security risk
  - Coupled with Python environment (general issues for renaming modules / changes)
- **Class definition**
  - Assume you trained a model with 2 layers FC1 and FC2
  - Now, you want a new model with 3 layers FC1, FC2 (from old model) and FC3
  - The loaded model won't see your code new changes. ONLY what is saved in pickle
- Saving just the **state_dict** allows you to easily **adapt the loaded** parameters to different architectures, as long as the tensor dimensions match
  - Troubles happen if there is a mismatch in keys' names
  - To load with missing keys:
  - model.load_state_dict(torch.load('**weights.pth**'), strict=**False**)

State keys

```python
class SimpleNet(nn.Module):
    def __init__(self, input_dim):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(input_dim, 30)
        self.fc2 = nn.Linear(30, 15)
        self.fc3 = nn.Linear(15, 1)
        self.activate = torch.relu

    def forward(self, x):
        x = self.fc1(x)
        x = self.activate(x)
        x = self.fc2(x)
        x = self.activate(x)
        x = self.fc3(x)
        return x
```

odict_keys(['fc1.weight', 'fc1.bias', 'fc2.weight', 'fc2.bias', 'fc3.weight', 'fc3.bias'])

```python
class SimpleNet(nn.Module):
    def __init__(self, input_dim):
        super(SimpleNet, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, 30),
            nn.ReLU(),
            nn.Linear(30, 15),
            nn.ReLU(),
            nn.Linear(15, 1)
        )

    def forward(self, x):
        return self.model(x)
```

odict_keys(['model.0.weight', 'model.0.bias', 'model.2.weight', 'model.2.bias', 'model.4.weight', 'model.4.bias'])

Checkpoints

- In deep learning, we typically save checkpoints to **resume** training later, perform model evaluation, or even ensemble models.
- A checkpoint usually includes model’s state, optimizer's state and other metadata.
- It is common to save several checkpoints
  - Sometimes we save the last X checkpoints (spaced with some factor)
  - Sometimes we save in every Y iterations, a checkpoint
  - In addition, we save the last model’s checkpoint

```
model = SimpleNet()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

epoch = 10
loss = 0.5

checkpoint = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
}

# Do some training and Save the checkpoint
torch.save(checkpoint, 'checkpoint.pth')
```

```python
# Initialize model and optimizer
model = SimpleNet()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

checkpoint = torch.load('checkpoint.pth')

model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])

epoch = checkpoint['epoch']
loss = checkpoint['loss']
```

Train-Evaluation Rounds

- If you have a single GPU, you typically train for X iterations, then evaluate once and so on
  - Downside, it takes more time to train a model
- If you have multiple GPUs, dedicate one of them for evaluation
  - This way the training pipeline keeps going

Which is the best model?

- Some people just count on the last trained model
- Others pick the best model during the training among the evaluation points

To GPU

- It is very important to make sure moving your model/data from the CPU to the GPU if you have one. Otherwise, you are just running on CPU
  - nvidia-smi can give you also a signal

```
cuda_available = torch.cuda.is_available()
device = torch.device("cuda:0" if cuda_available else "cpu")
model = model.to(device)
```

```
X_batch, y_batch = X_batch.to(device), y_batch.to(device)
```

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."