Facebook's Secure and Private AI Scholarship Challenge 2019

Notes by Pranjal Chaubey
https://www.linkedin.com/in/pranjallchaubey/
https://github.com/pranjalchaubey

UDACITY

Deep Learning with PyTorch

5th June 2019

Best methods for ~~matrix~~ tensor multiplication in PyTorch

torch.mm(tensor1, tensor2)

torch.matmul() — Broadcasts tensors, hence could give results even when the dimensions don't match. Not Recommended!

Methods to manipulate the shape of ~~tensor~~ tensor

weights.reshape(a,b) — 'Sometimes' copies the whole tensor to a new memory
— NOT EFFICIENT!

weights.resize_(a,b) — If the new shape doesn't match, it will add/subtract new elements in the tensor without giving any error.
— NOT RECOMMENDED!

weights.view(a,b) — Does what is asked, in the way it is expected.
— HIGHLY RECOMMENDED

.view(-1) — Flattens a tensor

images.view(images.shape[0], -1) — Flattens the 'image' tensor in this case

Size of the Batch

Loading the training data through DataLoader

```
trainloader = torch.utils.data.DataLoader
              (trainset, batch_size=64, shuffle=True)
```

Training Dataset — When iterating through 'trainloader', each iteration will contain a batch of 64 images.

Randomly pick images in a batch for training

Converting 'trainloader' in an iterable,

```
dataiter = iter(trainloader)
images, labels = dataiter.next()
```

Sigmoid Activation Function

```
def activation(x):
    return 1/(1+torch.exp(-x))
```

Softmax Function

```
def softmax(x)
    return torch.exp(x)/torch.sum(torch.exp(x),
                                  dim=1).view(-1, 1)
```

$$Softmax = \sigma(x_i) = \frac{e^{x_i}}{\sum_K^K e^{x_K}}$$

Building Networks with PyTorch nn Module

```
from torch import nn

class Network(nn.Module):
    def __init__(self):
        super().__init__()

        # Inputs to hidden layer linear transformation
        self.hidden = nn.Linear(784, 256)
        # Output layer
        self.output = nn.Linear(256, 10)

        # Define sigmoid activation & softmax output
        self.sigmoid = nn.Sigmoid()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        # Pass the input tensor through each operation
        x = self.hidden(x)
        x = self.sigmoid(x)
        x = self.output(x)
        x = self.softmax(x)

        return x
```

Registers the 'network' with nn module

This section can be removed using nn.functional

```
import torch.nn.functional as F
class Network(nn.module):
    def __init__(self):
        super().__init__()
        #Inputs to hidden layer linear transformations
        self.hidden = nn.Linear(784, 256)
        #Output Layer
        self.output = nn.Linear(256, 10)

    def forward(self, x):
        x = x.view(x.shape[0], -1)
        #Hidden Layer with sigmoid activation
        x = F.sigmoid(self.hidden(x))
        #Output Layer with Softmax
        x = F.softmax(self.output(x), dim=1)
        return x
```

Adding this will result in the tensor automatically getting flattened out by the network

A more succint version of the previous network

```
# Initialize the 'model' by creating the network object
model = Network()
```

Using nn.Sequential

```
# Hyperparameters for our network
input_size = 784
hidden_sizes = [128, 64]
output_size = 10

from collections import OrderedDict

model = nn.Sequential(OrderedDict([
    ('fc1', nn.Linear(input_size, hidden_sizes[0])),
    ('relu1', nn.ReLU()),
    ('fc2', nn.Linear(hidden_sizes[0], hidden_sizes[1])),
    ('relu2', nn.ReLU()),
    ('output', nn.Linear(hidden_sizes[1], output_size)),
    ('softmax', nn.Softmax(dim=1))]))
```

Layer names have to be UNIQUE

Ultra fast method to define a network

```
model.fc1.weight
model.fc1.bias
```
} Accessing various model layers

```
model.fc1.bias.data.fill_(0)
```
} Setting all biases to 0

```
model.fc1.weight.data.normal_(std=0.01)
```
↳ Setting weights to a sample from random normal distribution with a Standard Deviation of 0.01

Training a Neural Network

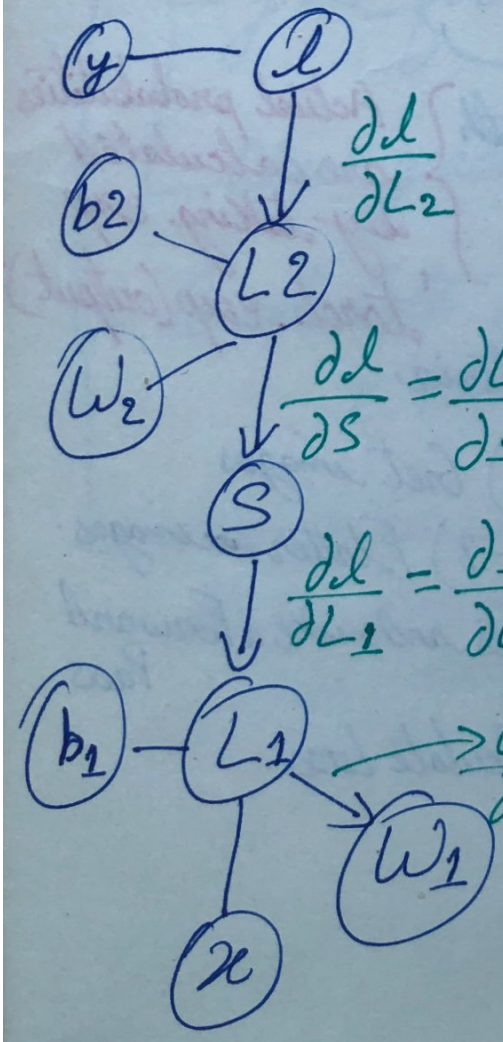
7th June 2019

We need a loss function to determine the errors our neural network is making, in case of regression and binary classification problems, often Mean Squared Error is used.

$$l = \frac{1}{2n} \sum_{i}^{n} (y_i - \hat{y}_i)^2$$

- $\frac{1}{2n}$: Number of examples
- y_i: True Labels
- $\hat{y}_i$: Predicted Labels

Backpropagation



We are effectively trying to figure out how 'l' is changing with respect to a change in 'W1'. Once we know that, we can minimize 'l' by updating weights,

$$W_1' = W_1 - \alpha \frac{\partial l}{\partial W_1}$$

Learning Rate

$$\frac{\partial l}{\partial L_2}$$

$$\frac{\partial l}{\partial S} = \frac{\partial L_2}{\partial S} \frac{\partial l}{\partial L_2}$$

$$\frac{\partial l}{\partial L_1} = \frac{\partial S}{\partial L_1} \frac{\partial l}{\partial S} = \frac{\partial S}{\partial L_1} \frac{\partial L_2}{\partial S} \frac{\partial l}{\partial L_2}$$

$$\frac{\partial l}{\partial W_1} = \frac{\partial L_1}{\partial W_1} \frac{\partial l}{\partial L_1}$$

$$= \frac{\partial L_1}{\partial W_1} \frac{\partial S}{\partial L_1} \frac{\partial L_2}{\partial S} \frac{\partial l}{\partial L_2}$$

Gradient

We typically use the nn.CrossEntropyLoss in PyTorch in classification problems.

Conventionally, the loss is assigned to a variable called criterion.

CrossEntropyLoss requires LOGITS as its inputs instead of the probabilities from the Softmax.

└ Logits are nothing but the INPUTS to the Softmax function.

CrossEntropyLoss is a combination of

nn.NLLLoss — Negative Log Likelihood Loss

nn.LogSoftmax

Used along with LogSoftmax

Actual probabilities are calculated by taking exp. 'torch.exp(output)'

```
criterion = nn.NLLLoss()                    Loss criteria
images, labels = next(iter(trainloader))    Get images
images = images.view(images.shape[0], -1)   Flatten the images
logits = model(images)                      Get the logits and make a Forward Pass
loss = criterion(logits, labels)            Calculate loss
print(loss)
```

AUTOGRAD

Autograd automatically calculates the gradients.

- Set requires_grad = True for a tensor
- Can be turned off for a block of code

```
x = torch.zeroes(1, requires_grad = True)
with torch.no_grad() — Turns off autograd
    y = x*2
```

- Gradients can be globally turned on and off using torch.set_grad_enabled(True/False)
- Keeps track of all the operations performed on a tensor
  - grad_fn can be used to check the function that generated this variable
  - Check gradients for a tensor tensor.grad
- To calculate the gradient of z w.r.t. x

```
z.backward()
```

- When a new network is created in PyTorch, all the parameters are initialized with requires_grad = True

8th June 2019

OPTIMIZER

Once we get the gradients from Autograd
- We need to update the weights
  - performed by an optimizer

Optimizers in PyTorch live inside the optim package
- from torch import optim

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

- SGD: Stochastic Gradient Descent
- model.parameters(): Model parameters to optimize
- lr: Learning Rate

General Steps to train a network in PyTorch
- Zero out the gradients since they keep getting accumulated with each batch
  - optimizer.zero_grad()
- Make a Forward Pass through the network
- Use the network output to calculate the loss
- Perform backward pass using loss.backward() to calculate the gradients
- Take a STEP with the optimizer to update the weights
- One pass through the entire dataset is called an EPOCH

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

images, labels = next(iter(trainloader))
images.resize_(64, 784)

optimizer.zero_grad()
output = model.forward(images)
loss = criterion(output, labels)
loss.backward()
optimizer.step()
```

```
epochs = 5
for e in range(epochs):
    running_loss = 0
    for images, labels in trainloader:
        images = images.view(images.shape[0], -1)

        optimizer.zero_grad()
        output = model.forward(images)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    else:
        print(f"Training loss {running_loss/
                               len(trainloader)}")
```

TRAINING LOOP

10th June 2019

VALIDATION

After training the model on 'training data', we need to 'validate' it on 'testing data' in order to check its accuracy.

Training data is loaded in trainloader.
Testing data is loaded in testloader.

Finding the model accuracy on a test batch is simple,

```
#Get the class probabilities
ps = torch.exp(model.forward(images))
#Use ps.topk(n) method that gives the highest
  'n' classes predicted for an input
# ps.topk(1) gives the highest class with the highest
  probability
top_p, top_class = ps.topk(1, dim = 1)
#Get the correctly predicted labels as 1
equals = top_class == labels.view(*top_class)
```

MARK

equals ← Byte Tensor

*top_class → To ensure correct dimensions of tensors

```
#Get the accuracy
accuracy = torch.mean(equals.type(torch.FloatTensor))
```

equals.type(torch.FloatTensor) → Convert to float to calculate mean

```
for e in range(epochs):
    for images, labels in trainloader:
        .
        .
        .
    else:
        test_loss = 0
        accuracy = 0
        with torch.no_grad():
        for images, labels in testloader:
            log_ps = model.forward(images)
            test_loss += criterion(log_ps, labels)

            ps = torch.exp(log_ps)
            top_p, top_class = ps.topk(1, dim=1)
            equals = top_class == labels.view(*top_class)
            accuracy += torch.mean(equals.type(torch.
                                   FloatTensor))

        print(f'Test loss: {test_loss/len(testloader)}')
        print(f'Accuracy: {accuracy/len(testloader)}')
```

Training loop

Turn off gradients to speed things up

VALIDATION Loop

OVERFITTING

As the network tends to learn the training data better and better, it tends to OVERFIT.

- As a result, its accuracy on the VALIDATION/TESTING DATA goes DOWN!
- To avoid overfitting
  - Early Stopping — Save different models and choose the one with lowest validation loss.
  - Dropout — Randomly drop neural units to make the network more robust.

```
self.dropout = nn.Dropout(p=0.2)
                          Drop Probability
x = self.dropout(F.relu(self.fc1(x)))
```

- During testing/validation we would want to use the entire model without dropout.
  - model.eval()
    - Back to training mode
      - model.train()

Saving and Loading Models

The model parameters for PyTorch networks are stored in a model's state_dict.

- `print(model.state_dict().keys())`
- Saving the models is easy
  - `torch.save(model.state_dict(), 'filename.pth')`
    - 'pth' — Extension for PyTorch Models
- Loading the state_dict
  - `state_dict = torch.load('filename.pth')`
  - Attaching the state_dict to a new model
    - `model.load_state_dict(state_dict)`

The new model needs to have exactly the same number of layers and the number of neurons per layer as the original model. Otherwise, PyTorch will throw an error.

Loading Image Data

Simplest way to load image data in PyTorch is through datasets.ImageFolder from torchvision

```
dataset = datasets.ImageFolder('path/to/data',
                               transform=transform)
```

The system expects the folder to have a particular structure,

```
root/dog/d1.png
root/dog/d2.png
root/cat/c1.png
root/cat/c2.png
   ⋮
```

Images are also usually transformed when they are loaded

```
transform = transforms.Compose([
        transforms.Resize(255),
        transforms.CenterCrop(224),
        transforms.ToTensor()])
```

Transforms Pipeline

We generally introduce randomness into the data used for training the network to make it more robust

- RandomRotation(30)
- RandomResizedCrop(224)
- RandomHorizontalFlip()
- Normalize([0.5,0.5,0.5], [0.5,0.5,0.5])
  - Keeps the network weights near 0
  - Makes backpropagation more stable

TRANSFER LEARNING

We use pre-trained models to solve hard Computer Vision problems

- from torchvision import datasets, transforms, models
  - imports pre-trained models
- Loading the models `model = models.densenet121(pretrained=True)`
- Need to freeze the model parameters so that we don't backpropagate through them

```
for param in model.parameters():
    param.requires_grad = False
```

- We only modify the final classifier layer according to our applications

```
classifier = nn.Sequential(
    ⋮
)
```

New layers that we will attach as the final layers in the pretrained network

```
model.classifier = classifier
```

Attach the new layers to the pretrained model

- Moving tensors back and forth from GPU to CPU

```
model.cuda()      model.cpu()
images.cuda()     images.cpu()
print(torch.cuda.is_available())
model.to('cuda')
model.to('cpu')
```

Check if cuda is available