CMSC 491/691: Computer Vision **Spring 2024**

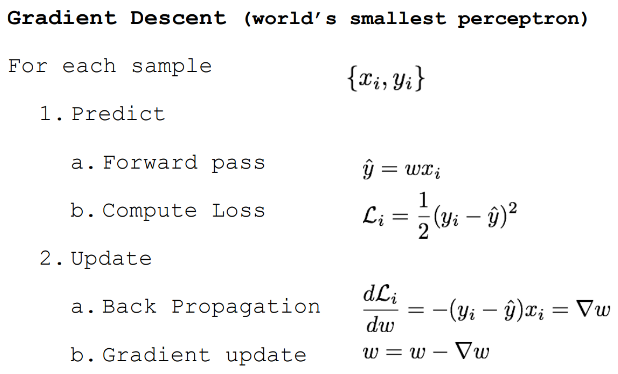
Lecture 10: 2024-03-11 Pytorch Tutorial

Lecturer: Tejas Gokhale *Scribe: Faisal Rasheed Khan*

The notes below are an example of what I am expecting. They were taken from a random graduate class. They illustrate some uses of various LATEX macros. Take a look at this and imitate.

10.1 Recap of Last Lecture

In the last lecture, we have seen the neural network optimization using gradient descent. The gradient descent aims to find atleast the local minima. The work is done with the help of the loss function and its derivative, where the weights are updated in the direction of negative gradients.
As seen in the previous chapters, we have seen various loss functions such as Mean-squared, Hinge loss to mention the few. For multi-layered network there are weights at each network, so we apply gradients and



```
Gradient Descent (world's smallest perceptron)
For each sample              ${x_i, y_i}$
  1. Predict
    a. Forward pass          $\hat{y} = wx_i$
    b. Compute Loss          $\mathcal{L}_i = \frac{1}{2}(y_i - \hat{y})^2$
  2. Update
    a. Back Propagation      $\frac{d\mathcal{L}_i}{dw} = -(y_i - \hat{y})x_i = \nabla w$
    b. Gradient update       $w = w - \nabla w$
```

Figure 10.1: Neural Network with Gradient descent

update it using the chain rule of the derivatives. We have also seen Covolutional Neural Networks and some

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \left[\frac{\partial \mathcal{L}}{\partial w_3} \frac{\partial \mathcal{L}}{\partial w_2} \frac{\partial \mathcal{L}}{\partial w_1} \frac{\partial \mathcal{L}}{\partial b} \right]$$

Figure 10.2: Multilayer gradients computation

of the architecture. In the next chapter, we see how to code using pytorch the neural networks.

10.2 Tensor Initialization

Initialization for a tensor can be done in various ways. It can be done using lists, numpy, from another tensor and also by random initialization by specifying the size. First, we need to import torch.

```
import torch
```

Using Lists:

```
command - torch.tensor()

data = [[1,2],[3,4]]
tensor_data = torch.tensor(data)
```

We can check the shape, type, datatype of data using the command:

```
data.shape
type(tensor_data)
tensor.dtype
```

Using Numpy Arrays:

```
command - torch.from_numpy()

import numpy as np
np_array = np.array(data)
torch_numpy = torch.from_numpy(np_array)
```

From another tensor:

```
command - torch.ones_like()

torch_ones = torch.ones_like(tensor_data, dtype=torch.float)
```

Using Rand:

```
command - torch.rand()
tensor = torch.rand(3,4)
```

We can also use GPU for the tensor processing:

```
if torch.cuda.is_available():
tensor = tensor.to("cuda")
```

10.3 Operations on Tensor

There are many operations which can be done on the tensor like indexing, slicing, concatenate tensors and arithmetic operations.

10.3.1 Indexing

For 1D tensor:

```
tensor[0] gives the first element
```

For Multi-dimensional tensors:

```
tensor[0] gives the first row
tensor[0][0] gives the first row first element
```

10.3.2 Slicing

For 1D tensor:

```
tensor[:] gives the all the elements
tensor[-1] gives the last element
tensor[::2] gives alternate elements which takes step 2
```

For Multi-dimensional tensors:

```
tensor[:,0] gives the first column
```

10.3.3 Arithmetic operations

torch.cat() concatenates multiple tensors and takes dim as an argument.
torch.stack() creates multidimensional tensors which also takes dim as an argument.
Theres element-wise multiplication which is given by tensor.mul(tensor2)
In order to do matrix multiplication we need to use tensor.matmul(tensor2). Make sure to have appropriate shape for the multiplication.

```
tensor[:] gives the all the elements
tensor[-1] gives the last element
tensor[::2] gives alternate elements which takes step 2
```

sectionLoading Datasets We can also load the dataset present in the various libraries of the torch. Need to specify the following commands for the packages:

```
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor

#Training
from torch.utils.data import DataLoader
train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)

#Iterate the training data
train_features, train_labels = next(iter(train_dataloader))
```

We can also transform the tensors using ToTensor.

10.4 Neural Network using Pytorch

```
import os
import torch
import torch.nn.functional as F
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5, 1, 0)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5, 1, 0)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = torch.flatten(x, -1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x


model = NeuralNetwork().to(device)
print(model)

model_parameters = filter(lambda p: p.requires_grad, model.parameters())

params = ([p.size() for p in model_parameters])


X = torch.rand(1,3, 32, 32, device=device)
```

```
logits = model(X)
pred_probab = nn.Softmax(dim=0)(logits)
print(logits)
y_pred = pred_probab.argmax(0)
print(pred_probab)
print(pred_probab.sum())
print(f"Predicted class: {y_pred}")
```

10.4.1 Gradients for the loss function (Single Layer)

```
x = torch.ones(5)  # input tensor
y = torch.zeros(3)  # expected output
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
z = torch.matmul(x, w)+b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
loss.backward()
print(w.grad)
print(b.grad)

#Disabling the gradient
z = torch.matmul(x, w)+b
print(z.requires_grad)

with torch.no_grad():
    z = torch.matmul(x, w)+b
print(z.requires_grad)
```

10.4.2 Batches for the network

```
loss_fn = nn.CrossEntropyLoss()

learning_rate = 1e-3
batch_size = 64
epochs = 5

optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)
        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

def test_loop(dataloader, model, loss_fn):
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")


loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

epochs = 10

for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
```

10.4.3 Save and Load the Model

```
model = models.vgg16(weights='IMAGENET1K_V1')
torch.save(model.state_dict(), 'model_weights.pth')

model = models.vgg16()
model.load_state_dict(torch.load('model_weights.pth'))
model.eval()
```