Optimization & Deep Learning Packages

Felix Kreuk & Yossi Adi

University of Bar-Ilan

May 15, 2018

No derivatives today!

No derivatives today!

Just kidding. Maybe a little bit.

Overview

1. Optimization
   - SGD
   - Momentum
   - AdaGrad
   - RMSprop
   - Adam

2. Deep Learning Packages
   - Tensors
   - Loading MNIST
   - First Model
   - Training & Testing

Outline

1. Optimization
   - SGD
   - Momentum
   - AdaGrad
   - RMSprop
   - Adam

2. Deep Learning Packages
   - Tensors
   - Loading MNIST
   - First Model
   - Training & Testing

Optimizing so far - SGD

- Recall our goal is $\theta = \arg\min \sum_{i=0}^{N} L(\theta, x, y)$

Optimizing so far - SGD

- Recall our goal is $\theta = \arg\min \sum_{i=0}^{N} L(\theta, x, y)$
- θ was found by SGD.

Optimizing so far - SGD

- Recall our goal is $\theta = \arg\min \sum_{i=0}^{N} L(\theta, x, y)$
- θ was found by SGD.
- $\theta_i \leftarrow \theta_i - \eta \cdot \frac{\partial L}{\partial \theta_i}$

Optimizing so far - SGD

- Recall our goal is $\theta = \arg\min \sum_{i=0}^{N} L(\theta, x, y)$
- θ was found by SGD.
- $\theta_i \leftarrow \theta_i - \eta \cdot \frac{\partial L}{\partial \theta_i}$

Challenges:

Optimizing so far - SGD

- Recall our goal is $\theta = \arg\min \sum_{i=0}^{N} L(\theta, x, y)$
- θ was found by SGD.
- $\theta_i \leftarrow \theta_i - \eta \cdot \frac{\partial L}{\partial \theta_i}$

Challenges:

- Choosing a proper learning rate can be difficult.

Optimizing so far - SGD

- Recall our goal is $\theta = \arg\min \sum_{i=0}^{N} L(\theta, x, y)$
- θ was found by SGD.
- $\theta_i \leftarrow \theta_i - \eta \cdot \frac{\partial L}{\partial \theta_i}$

Challenges:

- Choosing a proper learning rate can be difficult.
- Pre-defined learning rate schedules aren't adaptive.

Optimizing so far - SGD

- Recall our goal is $\theta = \arg\min \sum_{i=0}^{N} L(\theta, x, y)$
- θ was found by SGD.
- $\theta_i \leftarrow \theta_i - \eta \cdot \frac{\partial L}{\partial \theta_i}$

Challenges:

- Choosing a proper learning rate can be difficult.
- Pre-defined learning rate schedules aren't adaptive.
- The same learning rate for all parameters.

Optimizing so far - SGD

- Recall our goal is $\theta = \arg\min \sum_{i=0}^{N} L(\theta, x, y)$
- θ was found by SGD.
- $\theta_i \leftarrow \theta_i - \eta \cdot \frac{\partial L}{\partial \theta_i}$

Challenges:

- Choosing a proper learning rate can be difficult.
- Pre-defined learning rate schedules aren't adaptive.
- The same learning rate for all parameters.
- Escaping from saddle points and local minima is hard.

Optimizing so far - SGD

- Recall our goal is $\theta = \arg\min \sum_{i=0}^{N} L(\theta, x, y)$
- θ was found by SGD.
- $\theta_i \leftarrow \theta_i - \eta \cdot \frac{\partial L}{\partial \theta_i}$

Challenges:

- Choosing a proper learning rate can be difficult.
- Pre-defined learning rate schedules aren't adaptive.
- The same learning rate for all parameters.
- Escaping from saddle points and local minima is hard.

We outline some variants of Gradient Descent widely used by the deep learning community.

Outline

1. Optimization
   - SGD
   - Momentum
   - AdaGrad
   - RMSprop
   - Adam
2. Deep Learning Packages
   - Tensors
   - Loading MNIST
   - First Model
   - Training & Testing

Momentum

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum as in Image 2.

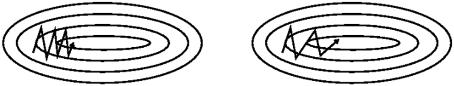


Figure: SGD with and without momentum

Momentum Update

$$v_t = \gamma v_{t-1} + \eta \frac{\partial L}{\partial \theta_i}$$

$$\theta_i = \theta_i - v_t$$

Outline

Adagrad

Why do all features have the same learning rate if they vary in importance and frequency seen?

Adagrad

Why do all features have the same learning rate if they vary in importance and frequency seen?

- Adagrad is an algorithm for gradient-based optimization that adapts the learning for each parameter.

Adagrad

Why do all features have the same learning rate if they vary in importance and frequency seen?

- Adagrad is an algorithm for gradient-based optimization that adapts the learning for each parameter.
- Weights that receive high gradients will have their effective learning rate reduced

Adagrad

Why do all features have the same learning rate if they vary in importance and frequency seen?

- Adagrad is an algorithm for gradient-based optimization that adapts the learning for each parameter.
- Weights that receive high gradients will have their effective learning rate reduced
- Weights that receive small or infrequent updates will have their effective learning rate increased

Adagrad

Why do all features have the same learning rate if they vary in importance and frequency seen?

- Adagrad is an algorithm for gradient-based optimization that adapts the learning for each parameter.
- Weights that receive high gradients will have their effective learning rate reduced
- Weights that receive small or infrequent updates will have their effective learning rate increased

AdaGrad Update

$$c_{t,i} = \left(\sum_{k=0}^{t} \frac{\partial L}{\partial \theta_{k,i}} \right)^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\frac{\partial L}{\partial \theta_{t,i}}}{\sqrt{c + \epsilon}}$$

Outline

1. Optimization
   - SGD
   - Momentum
   - AdaGrad
   - RMSprop
   - Adam

2. Deep Learning Packages
   - Tensors
   - Loading MNIST
   - First Model
   - Training & Testing

RMSprop

A downside of Adagrad is that in case of Deep Learning, the monotonic learning rate usually proves too aggressive and stops learning too early.

RMSprop

A downside of Adagrad is that in case of Deep Learning, the monotonic learning rate usually proves too aggressive and stops learning too early.

RMSprop Update

$$c_{t,i} = \gamma c_{t-1,i} + (1-\gamma)\left(\frac{\partial L}{\partial \theta_{t,i}}\right)^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\frac{\partial L}{\partial \theta_{t,i}}}{\sqrt{c+\epsilon}}$$

RMSprop

A downside of Adagrad is that in case of Deep Learning, the monotonic learning rate usually proves too aggressive and stops learning too early.

RMSprop Update

$$c_{t,i} = \gamma c_{t-1,i} + (1-\gamma)\left(\frac{\partial L}{\partial \theta_{t,i}}\right)^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\frac{\partial L}{\partial \theta_{t,i}}}{\sqrt{c+\epsilon}}$$

- Notice the update is the same as AdaGrad.

RMSprop

A downside of Adagrad is that in case of Deep Learning, the monotonic learning rate usually proves too aggressive and stops learning too early.

RMSprop Update

$$c_{t,i} = \gamma c_{t-1,i} + (1-\gamma)\left(\frac{\partial L}{\partial \theta_{t,i}}\right)^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\frac{\partial L}{\partial \theta_{t,i}}}{\sqrt{c+\epsilon}}$$

- Notice the update is the same as AdaGrad.
- Typical values for γ are 0.9, 0.99 and 0.999.

RMSprop

A downside of Adagrad is that in case of Deep Learning, the monotonic learning rate usually proves too aggressive and stops learning too early.

RMSprop Update

$$c_{t,i} = \gamma c_{t-1,i} + (1-\gamma)\left(\frac{\partial L}{\partial \theta_{t,i}}\right)^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\frac{\partial L}{\partial \theta_{t,i}}}{\sqrt{c+\epsilon}}$$

- Notice the update is the same as AdaGrad.
- Typical values for γ are 0.9, 0.99 and 0.999.
- Unlike AdaGrad, updates do not get monotonically smaller.

Outline

1. Optimization
   - SGD
   - Momentum
   - AdaGrad
   - RMSprop
   - Adam
2. Deep Learning Packages
   - Tensors
   - Loading MNIST
   - First Model
   - Training & Testing

Adam

OK, so each parameter is updates according to its frequency, but what about Momentum?

Adam

OK, so each parameter is updates according to its frequency, but what about Momentum?

Adam Update

$$m_{t,i} = \gamma_1 m_{t-1,i} + (1-\gamma_1)\frac{\partial L}{\partial \theta_{t,i}}$$

$$v_{t,i} = \gamma_2 v_{t-1,i} + (1-\gamma_2)\left(\frac{\partial L}{\partial \theta_{t,i}}\right)^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{m_{t,i}}{\sqrt{v_{t,i}+\epsilon}}$$

Adam

OK, so each parameter is updates according to its frequency, but what about Momentum?

Adam Update

$$m_{t,i} = \gamma_1 m_{t-1,i} + (1-\gamma_1)\frac{\partial L}{\partial \theta_{t,i}}$$

$$v_{t,i} = \gamma_2 v_{t-1,i} + (1-\gamma_2)\left(\frac{\partial L}{\partial \theta_{t,i}}\right)^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{m_{t,i}}{\sqrt{v_{t,i} + \epsilon}}$$

- Basically, Adam is a combination of Momentum and RMSprop.

Adam

OK, so each parameter is updates according to its frequency, but what about Momentum?

Adam Update

$$m_{t,i} = \gamma_1 m_{t-1,i} + (1-\gamma_1)\frac{\partial L}{\partial \theta_{t,i}}$$

$$v_{t,i} = \gamma_2 v_{t-1,i} + (1-\gamma_2)\left(\frac{\partial L}{\partial \theta_{t,i}}\right)^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{m_{t,i}}{\sqrt{v_{t,i}+\epsilon}}$$

- Basically, Adam is a combination of Momentum and RMSprop.
- Practically, it is the default algorithm in many packages.

Adam

OK, so each parameter is updates according to its frequency, but what about Momentum?

Adam Update

$$m_{t,i} = \gamma_1 m_{t-1,i} + (1-\gamma_1)\frac{\partial L}{\partial \theta_{t,i}}$$

$$v_{t,i} = \gamma_2 v_{t-1,i} + (1-\gamma_2)\left(\frac{\partial L}{\partial \theta_{t,i}}\right)^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{m_{t,i}}{\sqrt{v_{t,i}+\epsilon}}$$

- Basically, Adam is a combination of Momentum and RMSprop.
- Practically, it is the default algorithm in many packages.
- Recommended values in paper are $\gamma_1 = 0.9$, $\gamma_2 = 0.99$.

Visual Example

PyTorch

- In this course we will use **PyTorch**.
- PyTorch is a deep learning package by Facebook.
- Installation instructions can be found at https://pytorch.org

Outline

1. Optimization
   - SGD
   - Momentum
   - AdaGrad
   - RMSprop
   - Adam
2. Deep Learning Packages
   - Tensors
   - Loading MNIST
   - First Model
   - Training & Testing

Hello World - Tensors I

- Forget about vectors/matrices - we will use Tensors from now on.
- Tensors are n-dimensional vectors.
- Very similar to ndarrays in numpy.
- Used to represent data and parameters in PyTorch.

Hello World - Tensors II

Let's define 2 tensors:

```
>>> w = torch.FloatTensor([1,2,3])
>>> x = torch.FloatTensor([4,5,6])
```

We can multiply them element-wise

```
>>> w * x

  4
 10
 18
 [torch.FloatTensor of size 3]
```

Add:

```
>>> w + x

 5
 7
 9
[torch.FloatTensor of size 3]
```

Hello World - Tensors III

Concat:

```
>>> torch.cat([w,x])

  1
  2
  3
 15
  5
  6
[torch.FloatTensor of size 6]
```

Assign value:

```
>>> x[0] = 15
>>> x

 15
  5
  6
[torch.FloatTensor of size 3]
```

You can read more in the official documentation.

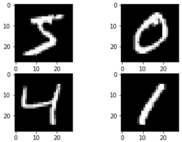
Outline

Hello World - Loading MNIST

First, we need to load the data. Fortunately, PyTorch makes it easy.

```
transforms = transforms.Compose([
                        transforms.ToTensor(),
                        transforms.Normalize((0.1307,), (0.3081,))])

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('./data', train=True, download=True,
           transform=transforms),
    batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('./data', train=False, transform=transforms),
    batch_size=64, shuffle=True)
```



Outline

1. Optimization
   - SGD
   - Momentum
   - AdaGrad
   - RMSprop
   - Adam
2. Deep Learning Packages
   - Tensors
   - Loading MNIST
   - First Model
   - Training & Testing

Hello World - Model

Let's create our first model, it will be a simple one.

```
class FirstNet(nn.Module):
        def __init__(self,image_size):
                super(FirstNet, self).__init__()
                self.image_size = image_size
                self.fc0 = nn.Linear(image_size, 1000)
                self.fc1 = nn.Linear(1000, 50)
                self.fc2 = nn.Linear(50, 10)

        def forward(self, x):
                x = x.view(-1, self.image_size)
                x = F.relu(self.fc0(x))
                x = F.relu(self.fc1(x))
                x = F.relu(self.fc2(x))
                return F.log_softmax(x)

model = FirstNet(image_size=28*28)
```

Note that we didn't have to define "backward" – it was already written for us.

Outline

1. Optimization
   - SGD
   - Momentum
   - AdaGrad
   - RMSprop
   - Adam
2. Deep Learning Packages
   - Tensors
   - Loading MNIST
   - First Model
   - Training & Testing

Hello World - Training Loop I

So now that we have our model, we can write a training loop for it:

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

def train(epoch, model):
    model.train()
    for batch_idx, (data, labels) in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, labels)
        loss.backward()
        optimizer.step()
```

- Loop through all training examples
- Calculate model's output
- Calculate loss
- Backprop the error
- Update model's parameters

Hello World - Training Loop II

You can swap SGD with any other optimizer:

```
optimizer = optim.Adam(model.parameters(), lr=lr)
optimizer = optim.AdaDelta(model.parameters(), lr=lr)
optimizer = optim.RMSprop(model.parameters(), lr=lr)
...
```

Hello World - Testing Loop

```
def test():
    model.eval()
    test_loss = 0
    correct = 0
    for data, target in test_loader:
        output = model(data)
        test_loss += F.nll_loss(output, target, size_average=False).data[0] # sum up batch loss
        pred = output.data.max(1, keepdim=True)[1] # get the index of the max log-probability
        correct += pred.eq(target.data.view_as(pred)).cpu().sum()

    test_loss /= len(test_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

- No weight updates
- Accumulating metrics (accuracy and average loss)

Putting it all together

```
for epoch in range(1, 10 + 1):
    train(epoch)
    test()
```

```
Train Epoch: 1 [0/60000 (0%)]   Loss: 2.308621
Train Epoch: 1 [6400/60000 (11%)]       Loss: 2.285088
Train Epoch: 1 [12800/60000 (21%)]      Loss: 2.263519
Train Epoch: 1 [19200/60000 (32%)]      Loss: 2.210358
Train Epoch: 1 [25600/60000 (43%)]      Loss: 2.217742
Train Epoch: 1 [32000/60000 (53%)]      Loss: 2.140791
Train Epoch: 1 [38400/60000 (64%)]      Loss: 2.171224
Train Epoch: 1 [44800/60000 (75%)]      Loss: 2.134961
Train Epoch: 1 [51200/60000 (85%)]      Loss: 2.020994
Train Epoch: 1 [57600/60000 (96%)]      Loss: 1.994740

Test set: Average loss: 2.0090, Accuracy: 4920/10000 (49%)
...
```