Report 9 - Deep Learning

May 31, 2020

1 Deep Learning with Pytorch

There are many deep learning frameworks available for machine learning in Python and the two most well-known are Tensorflow and PyTorch. An analysis of recent academic conferences has shown that the userbase of PyTorch in academia has overtaken that of TensorFlow — in conferences such as ICML and NeurIPS a *vast* majority of papers are now published with PyTorch code.

Due to the increasing use of PyTorch, we will demonstrate its use in this report.

In this report we will focus on the practical implementation of deep learning methods. We will discuss neural networks and how they are trained/optimized, and implement a deep image classification neural network with convolutional layers.

1.1 Implementing a Deep Neural Network

A deep neural network is simply a neural network with many hidden layers. As we will explore in this report, it was difficult to implement *deep* neural networks for a very long time due to numerical instability caused by vanishing and exploding gradients.

The first step in implementing a neural network is to define its **architecture**. Over time different architectures and layers of neural networks have become specialized to their applications. For example, in computer vision convolutional neural networks have become the norm as they provide the ability to detect edges, which may then allow the network to detect shapes, which may then allow the network to detect *objects*. Similarly, recurrent neural networks have become the norm in natural language processing as language is sequential: we write one word at the time, which depends on previous words.

In defining an architecture we are also defining the parameters of our network/model, which we want to learn. In order to train these parameters we input one observation from our dataset and make a prediction, this process is known as **forward propagation**. We then compute a loss. Our aim is to *minimize* this loss, and thus we compute the derivatives of the loss function with respect to *each* parameter in our model, and evaluate these derivatives for the given loss; this is known as **backpropagation**. We now have the parameters at the current iteration of training and the derivative of the loss w.r.t said parameters, so we can implement a simple gradient descent step using

$$\text{weight}_{i+1} = \text{weight}_i - \alpha \frac{\partial L}{\partial \text{weight}_i},$$

where weight_i is a parameter in the model, α is the learning rate, and L is the loss function. We can obtain much better performance from our model by replacing this gradient descent optimizer with others such as gradient descent with momentum, Adam, RMSprop, etc.

To summarize, in order to train a neural network we must: 1. Define a netural network architecture. 2. Iterate the following over a dataset of inputs: 1. Pass the input through the network and make a prediction. 2. Compute the loss. 3. Propagate the gradients backwards through the network. 4. Update the network's parameters.

Let's implement a simple neural network that we all know: a **linear model**. To do this we are going to make use of a layer typically used in the last few layers of many neural networks called a fully-connected layer. In this case our neural network will only have an input layer and an output layer. Suppose we observe ten covariates.

[24]:
```
import torch
import torch.nn as nn

class LM(nn.Module):

    def __init__(self):
        super(LM, self).__init__()
        # a fully connected layer mapping 10 inputs to 1 output
        self.fc1 = nn.Linear(10,1)

    def forward(self, x):
        # in the forward step we apply no transformation
        x = self.fc1(x)
        return x


linear_model = LM()
print(linear_model)
```

```
LM(
  (fc1): Linear(in_features=10, out_features=1, bias=True)
)
```

In the above code we have defined the layers of our neural network in `def __init__(self):`, and defined how to go from one layer to the next in `def forward(self, x):`. The backpropagation function is defined for us using PyTorch's `autograd` attribute for Tensor objects and operations.

We can easily access the model's parameters. Below we see that we have 11 parameters: the 10 coefficients corresponding to the 10 inputs, and the intercept (often called bias in machine learning).

[25]:
```
params = list(linear_model.parameters())
print(params)
print(params[0].size())
```

```
[Parameter containing:
tensor([[-0.0220,  0.2401, -0.1822, -0.0323, -0.2773, -0.0259,  0.1816,  0.1159,
          0.1656, -0.1080]], requires_grad=True), Parameter containing:
tensor([-0.0457], requires_grad=True)]
torch.Size([1, 10])
```

We can test that the model works by simply plugging in some random inputs.

[26]:
```
# print the output for one input
input = torch.randn(1,10)
out = linear_model(input)
print(out)
```

```
tensor([[0.2121]], grad_fn=<AddmmBackward>)
```

[27]:
```
# print the output for three inputs
input = torch.randn(3,10)
out = linear_model(input)
print(out)
```

```
tensor([[-0.4303],
        [-1.0122],
        [ 0.9595]], grad_fn=<AddmmBackward>)
```

We see that the model successfully outputs predictions. Let's use scikit-learn to generate data to test our model. Below we generate 100 observations of 10 feature variables and the response variable. Note that only 3 of the features have an impact on the response.

[40]:
```
import numpy as np
from sklearn import datasets

# generate dataset with 100 obs, 10 covariates, 3 influential covariates
dataset = datasets.make_regression(100,10,3)

# convert to pytorch tensor
X = torch.from_numpy(np.array(dataset[0]))
y = torch.from_numpy(np.array(dataset[1]))
```

Let's look at some of the generated dataset.

[29]:
```
# print first observation stored in X
print(X[0])
```

```
tensor([-1.2169,  1.7258, -0.2211,  0.1203,  0.1417, -0.4764, -0.0167,  0.6515,
         0.4356,  1.1793], dtype=torch.float64)
```

[30]:
```
# print first observed value of the response
print(y[0])
```

```
tensor(-67.4774, dtype=torch.float64)
```

In order to train the model on this dataset we must assign a loss function and an optimizer. We will use the mean-squared error loss function. For demonstrative purposes, let's compute the loss for a single observation.

[31]:
```
# convert lm weights to type double
linear_model.double()

# make prediction and assign the true response
prediction = linear_model(X[0])
target = y[0].view(-1) # ensure target has same dimension as input

# assign loss function
criterion = nn.MSELoss()
loss = criterion(prediction, target)

# print loss
print(loss)
```

```
tensor(4610.3635, dtype=torch.float64, grad_fn=<MseLossBackward>)
```

Our loss is very large, but we haven't trained the model yet. It has only been intialized by setting the weights randomly. Let's first train the weights using our own implementation of gradient descent (steepest descent) and then explore the optimizers offered in PyTorch.

In order to backpropagate the error all we need to do is call `loss.backward()`. You may have noticed that every Tensor we have printed provides us with its `grad_fn` attribute. This attribute stores the entire history of operations done on the Tensor and allows us to trace backwards through the computational graph. This attribute makes it possible to perform backpropagation.

Below we call `backward()` on the loss function. Note that we must always set the model's gradients to zero before doing this as the gradients will not be *replaced*, they will be *accumulated*.

[32]:
```
print(linear_model.fc1.weight)
```

```
Parameter containing:
tensor([[-0.0220,  0.2401, -0.1822, -0.0323, -0.2773, -0.0259,  0.1816,  0.1159,
          0.1656, -0.1080]], dtype=torch.float64, requires_grad=True)
```

[33]:
```
# set gradients to zero
linear_model.zero_grad()

# print gradients before calling backward
print(linear_model.fc1.weight.grad)

# backpropagate
loss.backward()

# print gradients
print(linear_model.fc1.weight.grad)
```

```
None
tensor([[-165.2549,  234.3588,  -30.0278,   16.3383,   19.2374,  -64.7016,
           -2.2617,   88.4716,   59.1506,  160.1506]], dtype=torch.float64)
```

After setting gradients equal to zero they are are actually set to `None` in order to support sparse gradients.

Let's implement a *very* simply parameter update using gradient descent.

[34]:
```
# print parameters before updating
print(list(linear_model.parameters()))

lr = 0.1
for f in linear_model.parameters():
    f.data.sub_(lr * f.grad.data)

# print parameters after updating
print(list(linear_model.parameters()))
```

```
[Parameter containing:
tensor([[-0.0220,  0.2401, -0.1822, -0.0323, -0.2773, -0.0259,  0.1816,  0.1159,
          0.1656, -0.1080]], dtype=torch.float64, requires_grad=True), Parameter
containing:
tensor([-0.0457], dtype=torch.float64, requires_grad=True)]
[Parameter containing:
tensor([[ 16.5035, -23.1958,   2.8205,  -1.6661,  -2.2010,   6.4442,   0.4078,
          -8.7313,  -5.7494, -16.1230]], dtype=torch.float64,
       requires_grad=True), Parameter containing:
tensor([-13.6256], dtype=torch.float64, requires_grad=True)]
```

PyTorch offers many sophisticated update regimes which you can find in `torch.optim`. An example of how to implement these optimizers can be found below.

[35]:
```
import torch.optim as optim

# assign optimizer
optimizer = optim.SGD(linear_model.parameters(), lr = 0.01)

# how to perform an update
optimizer.zero_grad()                   # zero the gradients
prediction = linear_model(X[0])         # make a prediction
target = y[0].view(-1)                  # assign target
loss = criterion(prediction, target)    # compute the loss in making a␣
 ↪prediction
loss.backward()                         # backpropagate
optimizer.step()                        # update parameters
```

Now let's combine what we have seen so far to train the model using the entire dataset.

[36]:
```
# re-initialize model
linear_model = LM()
linear_model.double()

# assign optimizer
optimizer = optim.SGD(linear_model.parameters(), lr = 0.003)
```

```python
for epoch in range(20):

    running_loss = 0.0
    for i, inputs in enumerate(X):
        # assign target
        target = y[i].view(-1)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward step -> backward step -> update parameters
        prediction = linear_model(inputs)
        loss = criterion(prediction, target)
        loss.backward()
        optimizer.step()

        # accumulate loss
        running_loss += loss.item()

    # print statistics at each epoch
    print(f"[epoch: {epoch+1}] loss: {running_loss/100}")
print("Finished training!")
```

```
[epoch: 1] loss: 10895.959484068162
[epoch: 2] loss: 3373.1894932040177
[epoch: 3] loss: 1237.905185006642
[epoch: 4] loss: 505.7780592095858
[epoch: 5] loss: 222.78215487654353
[epoch: 6] loss: 103.56622945514259
[epoch: 7] loss: 49.9977239906921
[epoch: 8] loss: 24.764884935843646
[epoch: 9] loss: 12.479095517540607
[epoch: 10] loss: 6.360651430374393
[epoch: 11] loss: 3.2670609550839567
[epoch: 12] loss: 1.6868959696324382
[epoch: 13] loss: 0.8741893172304885
[epoch: 14] loss: 0.4542117125670174
[epoch: 15] loss: 0.2364551742910063
[epoch: 16] loss: 0.12327545424854891
[epoch: 17] loss: 0.06434342985289072
[epoch: 18] loss: 0.033615144970956
[epoch: 19] loss: 0.017575157097415257
[epoch: 20] loss: 0.00919485931998835
Finished training!
```

Let's print the model parameters to see if the model correctly identified that only three coefficients should be non-zero (recall that the response for the synthetic data only depends on 3 of the

10 features).

[37]:
```
list(linear_model.parameters())
```

[37]:
```
[Parameter containing:
 tensor([[ 9.2244e+01, -2.5593e-02, -3.5501e-02, -1.8276e-02,  5.6669e+01,
           3.9044e-02, -7.9005e-03, -1.4960e-02,  8.4568e+01, -2.4409e-04]],
        dtype=torch.float64, requires_grad=True), Parameter containing:
 tensor([-0.0393], dtype=torch.float64, requires_grad=True)]
```

The model seems to have done well! Most of the model parameters are very close to zero with only three which are clearly non-zero.

1.2 Training a neural network

Now that we have implemented the most simple neural network possible (linear regression) in PyTorch, let's simulate some data which *cannot* be adequately modeled with a linear model and see how a neural network with two or more layers performs.

Recall that linear regression can only model linear combinations of our input variables, and so an easy way to simulate data which a linear model cannot represent may be to combine non-linear combinations of our input variables. The true power of neural networks stems from their ability to learn highly-complex functions of our input variables.

[42]:
```
# generate x values
x1 = np.random.normal(-3, 1, 10000)
x2 = np.random.normal(10, 1, 10000)
x = torch.from_numpy(np.array(list(zip(x1, x2)))) # combine x1 and x2 into a␣
↪model matrix
y = x[:,0] ** 2 + 14*x[:,1] ** 3                                                ␣
↪                                                                               ␣
↪                                                                               ␣
↪                                                                               ␣
↪                                                                               ␣
↪                                                                               ␣
↪                                                                               ␣
↪                                                                               ␣
↪                                                                               ␣
↪                                                                               ␣
↪                                                                               ␣
↪                                                                               ␣
↪                                                                               ␣
↪                                         + 3                         #␣
↪non-linear combination of x1 and x2
```

Our data is simulated as follows:$y = X_1^2 + 14X_2^3$, where$X_1 \sim N(-3,1)$ and$X_2 \sim N(10,1)$. We have generated 10000 observations.

We now attempt to fit a linear model to the data.

[43]:
```
# change regression model to take a user-defined number of inputs
class LM(nn.Module):
```

```
    def __init__(self, n_inputs):
        super(LM, self).__init__()
        # a fully connected layer mapping 10 inputs to 1 output
        self.fc1 = nn.Linear(n_inputs,1)

    def forward(self, x):
        # in the forward step we apply no transformation
        x = self.fc1(x)
        return x

# construct a linear model with 2 input variables
linear_model = LM(2).double()

# choose loss function
criterion = nn.MSELoss()

# choose optimizer
optimizer = optim.SGD(linear_model.parameters(), lr=0.003)

# train the model
for epoch in range(15):

    running_loss = 0.0
    for i, inputs in enumerate(x):
        # assign target
        target = y[i].view(-1)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward step -> backward step -> update parameters
        prediction = linear_model(inputs)
        loss = criterion(prediction, target)
        loss.backward()
        optimizer.step()

        # accumulate loss
        running_loss += loss.item()
    # print statistics
    print(f"[epoch: {epoch+1}] loss: {running_loss/10000:.2f}")

print("Finished training!")
```

```
[epoch: 1] loss: 7713554.22
[epoch: 2] loss: 3004742.73
[epoch: 3] loss: 1436064.96
[epoch: 4] loss: 899066.25
```

```
[epoch: 5] loss: 710681.77
[epoch: 6] loss: 642073.87
[epoch: 7] loss: 615726.53
[epoch: 8] loss: 604900.64
[epoch: 9] loss: 600103.30
[epoch: 10] loss: 597816.51
[epoch: 11] loss: 596657.68
[epoch: 12] loss: 596043.11
[epoch: 13] loss: 595706.97
[epoch: 14] loss: 595519.47
[epoch: 15] loss: 595413.62
Finished training!
```

We see that after training the model for more than ten epochs, the loss seems to have stalled and no longer decreases towards zero. This is because a linear model simply cannot adequately represent the true nature of the response.

Let's try to improve the performance of our predictive model by increasing the number of hidden neurons and layers. Below our model has two hidden layers with 64 and 32 neurons respectively. Note that in this case we have 64×32 weights to learn rather than 2 and so we have to implement mini-batch optimization, and a more sophisticated optimization method Adam which is explained below. We also have to train the model for *much* longer.

[47]:
```
import torch.nn.functional as F

# change regression model to take a user-defined number of inputs
class net(nn.Module):

    def __init__(self, n_inputs):
        super(net, self).__init__()
        # a fully connected layer mapping 10 inputs to 1 output
        self.fc1 = nn.Linear(n_inputs, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 1)

    def forward(self, x):
        # in the forward step we apply no transformation
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# construct a linear model with 2 input variables
linear_model = net(2).double()

# choose loss function
criterion = nn.MSELoss()

# choose optimizer
```

```
optimizer = optim.Adam(linear_model.parameters())

# train the model
batch_size = 32
for epoch in range(1000):

    batch_loss = 0.0
    permutation = torch.randperm(x.size()[0])

    for i in range(0, x.size()[0], batch_size):
        # assign target
        idx = permutation[i:i+batch_size]
        batch_x = x[idx]
        batch_y = y[idx].view(-1,1)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward step -> backward step -> update parameters
        prediction = linear_model(batch_x)
        loss = criterion(prediction, batch_y)
        loss.backward()
        optimizer.step()

        # accumulate loss
        batch_loss += loss.item() * batch_size

    # print statistics at each epoch
    if epoch % 100 == 99:
        print(f"[epoch: {epoch+1}] loss: {batch_loss/(X.size()[0]/batch_size):.
↪2f}")
print("Finished training!")
```

```
[epoch: 100] loss: 14687132.23
[epoch: 200] loss: 544259.51
[epoch: 300] loss: 351897.06
[epoch: 400] loss: 225667.19
[epoch: 500] loss: 245371.16
[epoch: 600] loss: 116662.77
[epoch: 700] loss: 193235.01
[epoch: 800] loss: 157885.21
[epoch: 900] loss: 117024.21
[epoch: 1000] loss: 124035.55
Finished training!
```

We have obtained a much lower loss and therefore better predictive performance than that obtained from the simple linear model. However, we have far more parameters to learn and so we had to train for far more epochs! The greater flexibility offered by the neural network comes

at a great computational cost. The network implemented above had a few wide layers (wide in that they have a large number of hidden units) but we may have obtained better results by implementing a deep but narrow network. When implementing a neural network it's important to experiment with the number of layers and their width.

1.2.1 Optimization methods

Let's quickly discuss optimization methods for machine learning. Or rather, ways of implementing an optimization method.

When implementing an optimization algorithm such as gradient descent, we have three options 1. Batch gradient descent — We produce predictions for the entire dataset, compute the loss, and backpropagate the errors. In order to perform one parameter update we must perform a pass through the entire dataset. One pass through the dataset is referred to as an *epoch*. This method is optimal in that the loss will *always* decrease. 2. Mini-batch gradient descent — We divide the dataset into mini-batches and produce predictions for one mini-batch at a time, compute the loss, backpropagate the error, and update the parameters. In doing this we can update the parameters *multiple* times per epoch and thus we can quickly decrease the error after initialization. Due to the randomness in choosing a mini-batch of data, the loss will *not* always decrease after updating the parameters. Popular choices of batch size are 32, 64, and sometimes 128. Given a fixed amount of computation time we may obtain better results by using this method. 3. Stochastic gradient descent (SGD) — We make predictions on a single observation in our dataset, compute the loss, backpropagate the gradients, and update the parameters. This is effectively mini-batch gradient descent with a batch size of 1. If we have n observatons then we perform n parameter updates per epoch. Once again, as we randomly select one observationa t a time and update the parameters, the loss may not necessarily decrease.

Let's plot the loss function for the three different approaches. We use a more narrow neural network in order to avoid training for a large number of epochs.

[403]:
```
import matplotlib.pyplot as plt

# change regression model to take a user-defined number of inputs
class net(nn.Module):

    def __init__(self, n_inputs):
        super(net, self).__init__()
        # a fully connected layer mapping 10 inputs to 1 output
        self.fc1 = nn.Linear(n_inputs, 16)
        self.fc2 = nn.Linear(16, 8)
        self.fc3 = nn.Linear(8, 1)

    def forward(self, x):
        # in the forward step we apply no transformation
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# construct a linear model with 2 input variables
```

```
linear_model = net(2).double()

# choose loss function
criterion = nn.MSELoss()

# choose optimizer
optimizer = optim.Adam(linear_model.parameters())

## SGD
loss_sgd = []
# train the model
batch_size = 1
for epoch in range(10):

    batch_loss = 0.0
    permutation = torch.randperm(x.size()[0])

    for i in range(0, x.size()[0], batch_size):
        # assign target
        idx = permutation[i:i+batch_size]
        batch_x = x[idx]
        batch_y = y[idx].view(-1,1)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward step -> backward step -> update parameters
        prediction = linear_model(batch_x)
        loss = criterion(prediction, batch_y)
        loss.backward()
        optimizer.step()

        # accumulate loss
        batch_loss += loss.item() * batch_size
        loss_sgd.append(loss.item())
print("Finished training!")

# construct a linear model with 2 input variables
linear_model = net(2).double()

# choose loss function
criterion = nn.MSELoss()

# choose optimizer
optimizer = optim.Adam(linear_model.parameters())

## mini-batch
```

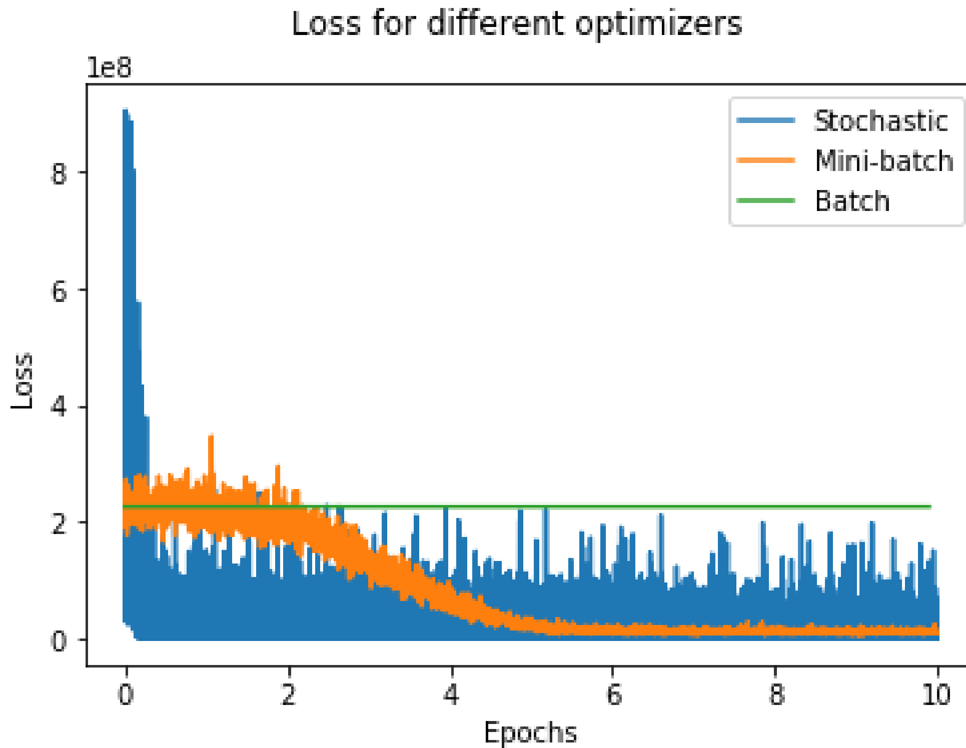
```
loss_mb = []
# train the model
batch_size = 32
for epoch in range(10):

    batch_loss = 0.0
    permutation = torch.randperm(x.size()[0])

    for i in range(0, x.size()[0], batch_size):
        # assign target
        idx = permutation[i:i+batch_size]
        batch_x = x[idx]
        batch_y = y[idx].view(-1,1)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward step -> backward step -> update parameters
        prediction = linear_model(batch_x)
        loss = criterion(prediction, batch_y)
        loss.backward()
        optimizer.step()

        # accumulate loss
        batch_loss += loss.item() * batch_size
        loss_mb.append(loss.item())
print("Finished training!")

# construct a linear model with 2 input variables
linear_model = net(2).double()

# choose loss function
criterion = nn.MSELoss()

# choose optimizer
optimizer = optim.Adam(linear_model.parameters())

## batch
loss_b = []
# train the model
batch_size = 10000
for epoch in range(10):

    batch_loss = 0.0
    permutation = torch.randperm(x.size()[0])

    for i in range(0, x.size()[0], batch_size):
```

```python
        # assign target
        idx = permutation[i:i+batch_size]
        batch_x = x[idx]
        batch_y = y[idx].view(-1,1)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward step -> backward step -> update parameters
        prediction = linear_model(batch_x)
        loss = criterion(prediction, batch_y)
        loss.backward()
        optimizer.step()

        # accumulate loss
        batch_loss += loss.item() * batch_size
        loss_b.append(loss.item())
print("Finished training!")
```

```
Finished training!
Finished training!
Finished training!
```

[463]:
```python
# plot losses
#%matplotlib inline
fig = plt.figure()
plt.plot(np.arange(0,10,1e-4),loss_sgd, label="Stochastic")
plt.plot(np.arange(0,10,10/3130),loss_mb, label="Mini-batch")
plt.plot(np.arange(0,11,11/10), loss_b, label = "Batch")
plt.legend()
plt.suptitle("Loss for different optimizers")
plt.xlabel("Epochs")
plt.ylabel("Loss")
```

[463]:
```
Text(0, 0.5, 'Loss')
```



As expected, SGD provides a rapid decrease in the training loss but also has an overwhelming amount of noise. Batch gradient descent look to be nearly constant, but only relative to the two other methods! It is in fact decreasing. Finally, mini-batch gradient descent with a batch size of 32 provides a nice balance — we get a quick decrease in loss and little noise relative to SGD.

2 Computer Vision

We now implement some more sophisticated architectures and perform classification on the well-known MNIST handwriting dataset. In this section we will combine what we have discussed previously with convolutional layers and data augmentation.

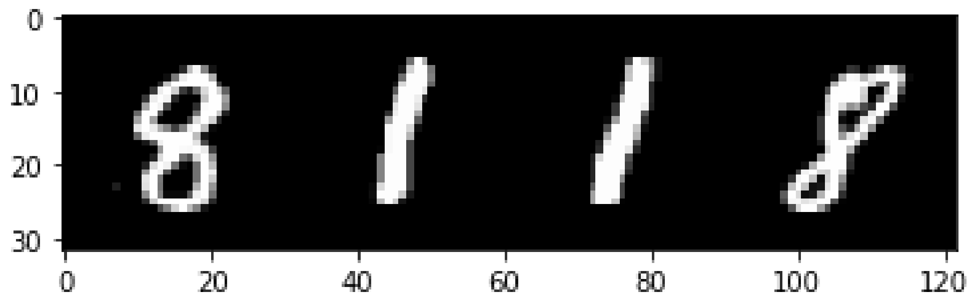
PyTorch makes it very easy to import well-known computer vision datasets using `torchvision`. Below we import the MNIST dataset, apply data augmentations (via `torchvision.transforms`) and split the data into a train and test set.

[50]:
```
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose([
    transforms.ToTensor()
])

trainset = torchvision.datasets.MNIST(root="./data", train=True,
                                      download=True, transform=transform)
```

```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=4)

testset = torchvision.datasets.MNIST(root="./data", train=False,
                                     download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=True, num_workers=4)
```

Below we define a generic function to show a batch of images.

[54]:
```
import matplotlib.pyplot as plt

#%matplotlib inline
def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

# get a batch of images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# plot
imshow(torchvision.utils.make_grid(images))
# print labels
print(f"labels: {[list(range(0,10))[j] for j in labels]}")
```




```
labels: [8, 1, 1, 8]
```

When we import the dataset using `torchvision.datasets` we can choose to apply transforms. In machine learning the process of transforming data is known as data augmentation, and the reasoning is simple: given a finite set of data we can increase our number of samples by applying reasonable transforms to the data. For example, suppose we want to train a classifier to identify dogs but we only have one hundred images. Rather than go out and take more photographs of dogs, we can double the size of our dataset by simply considering the horizontal reflection of all images. We could also apply rotations to our images, adjust the hue and saturation, etc. All of these transforms are easily applied using `torchvision.transforms`.

After loading the dataset as `trainset` and `testset` we also use PyTorch data loaders to combine the datasets with a sampler. The `DataLoader` provides an iterable which allows us to easily load samples in parallel.

We will now define the network architecture. We will implement a convolutional neural network (CNN) which is very similar to the neural networks introduced above, except they explicitly assume that the input is an image. These networks still have learnable weights, compute predictions, and provide a differentiable loss function allowing us to perform backpropagation. They simply differ in the types of layers used.

We cannot use the fully-connected layers seen previously for images as they do not scale well. For the MNIST dataset we have greyscale images of size 28x28, and so we have 28x28=784 feature variables. In most modern computer vision tasks we have colour images and so our input images has three variables per pixel (RGB). For a small 200x200 picture we would have 200x200x3=120,000 inputs! Training a model with fully-connected layers mapping from 120,000 input parameters will be extremely computationally expensive, and in order to avoid overfitting we would need an *extremely* large dataset.

Convolutional neural networks overcome this problem by applying filters to the input using convolutional kernels. The neural networks introduced previously map a list of inputs to a list of neurons, and so on. When the input is an image the layers in our network have another dimension: *depth*. For the CIFAR10 dataset, images have dimension 32x32x3 and so we think of layers in the network as *3D volumes*. When we implement convolutional layers to the network we also have to define their depth.

A possible network architecture for a simple CNN could be [INPUT → CONV → RELU → POOL → FC]. These layers can be explained in more detail:

1. INPUT: For the MNIST dataset our images are of size 28x28x1.

2. CONV: The convolutional layer will allow us to extract features from the input by applying a learnable filter to each layer of the input. For this layer we must specify its size(width=height), stride, and the number of filters (this defines the depth of the CONV layer). The CONV layer may correspond to a 28x28x6 volume. The intuition of CONV layers is explained below.

3. RELU: Applies an element-wise activation function. Typically we use ReLU which sets $x^* = max(0, x)$.

4. POOL: Pooling layers allow us to downsample an input without losing much information. Popular pooling methods include max pooling and average pooling. If we implement max pooling with size 2x2 then for 4 input neurons we will only retain the maximum value, reducing the 28x28x6 input dimension to one of size 14x14x6.

5. FC: This layer maps its input volume to a volume containing scores describing how likely each output label is (relative to one another). For the MNIST problem our output will be 1x1x10.

2.1 Convolutional Layer

The convolutional layer is the most important layer in the convolutional neural network. As stated above, the convolutional layer consists of learnable filters which we slide over the data. For example for the first layer on our MNIST CNN we may choose a convolutional layer of size

5x5x1. During the forward pass we slide the filter over the entire input volume and compute the dot products between entries in the filter and the pixel values. This outputs a matrix containing the reponse of the filter for each spatial position. The intuition behind convolutional layers (and this can be shown in practice) is that the layer can detect visual features such as edges or certain patterns. Later layers in the network take the response of these convolutional layers and input, and combine them. For example some neurons in later layers may combine edges to detect a circle, others may detect windows, others may detect a door, and then later layers combine these features to detect a car. The large number of neurons in a CNN often make it difficult to interpret the underlying model, but papers have attempted to show what layers in CNNs are actually detecting.

In implementing a convolutional layer there are a number of hyperparameters to choose.

1. Stride: In the above we mentioned that we slide/convolve the filter over the input volume. The *stride* determines the number of pixels we slide over. For example if the stride is 1 then we slide the filter to the next row of pixels, and if the stride is 2 then we skip 1 row of pixels and slide the filter to the 2nd row of pixels. The stride is often kept at 1 or 2 and is only really used to reduce the dimension of the output volume, but there are better ways to achieve this such as via pooling.
2. Padding: Padding determines the border of zeros we place around the image. Without doing this we would not be able to apply the filter with the filter centered at the edge of the input volume. The amount of padding is again a parameter to tune, but is often chosen to ensure that the layer's output dimension matches the input dimension. A padding value of 1 indicates that a border of zeros will be placed around the image, of width 1 pixel.
3. Depth: This determines the number of filters which we apply. We use more than one filter as one filter may find vertical edges, another may find horizontal edges, another may find patterns of colors. Typically the depth increases for the first few convolutional layers whilst the spatial dimensions (layer height and width) decrease.

2.1.1 MNIST Example

We now continue with the MNIST example, using the ideas discussed above.

[55]:
```
# define network architecture
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 4, 5) # output is 24x24x4
        self.pool1 = nn.MaxPool2d(2, 2) # output is 12x12x4
        self.conv2 = nn.Conv2d(4, 8, 5) # output is 8x8x8
        self.pool2 = nn.MaxPool2d(2, 2) # output is 4x4x8
        self.fc1 = nn.Linear(8*4*4, 32)
        self.fc2 = nn.Linear(32, 16)
        self.fc3 = nn.Linear(16, 10)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = x.view(-1, 8*4*4)
        x = F.relu(self.fc1(x))
```

```
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# create model
net = CNN()
print(net)
```

```
CNN(
  (conv1): Conv2d(1, 4, kernel_size=(5, 5), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv2): Conv2d(4, 8, kernel_size=(5, 5), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (fc1): Linear(in_features=128, out_features=32, bias=True)
  (fc2): Linear(in_features=32, out_features=16, bias=True)
  (fc3): Linear(in_features=16, out_features=10, bias=True)
)
```

The above architecture can be summarized as follows:

[INPUT → CONV → RELU → POOL → CONV → RELU → POOL → FC → RELU → FC → RELU → FC]

I have placed comments in the code to make it clear what the dimensions are of the output of each layer. To determine the spatial dimensions of convolutional layer outputs we use the following formula

$$H_{\text{out}} = \frac{H_{\text{in}} - F_h + 2P}{S} + 1,$$

where H_{in} and H_{out} denote the height of the input and output respectively, F_h denotes the height of the filter, P denotes the amount of padding, and S denotes the stride. We can similarly write down the formula for the width of the output

$$W_{\text{out}} = \frac{W_{\text{in}} - F_w + 2P}{S} + 1.$$

We now train the model on the MNIST dataset.

[56]:
```
# assign loss function
criterion = nn.CrossEntropyLoss()

# assign optimizer
optimizer = optim.SGD(net.parameters(), lr=0.001)

# train the network
for epoch in range(3):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
```

```python
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

```
[1,  2000] loss: 2.311
[1,  4000] loss: 2.307
[1,  6000] loss: 2.302
[1,  8000] loss: 2.299
[1, 10000] loss: 2.293
[1, 12000] loss: 2.287
[1, 14000] loss: 2.266
[2,  2000] loss: 2.079
[2,  4000] loss: 1.309
[2,  6000] loss: 0.706
[2,  8000] loss: 0.505
[2, 10000] loss: 0.404
[2, 12000] loss: 0.348
[2, 14000] loss: 0.325
[3,  2000] loss: 0.285
[3,  4000] loss: 0.254
[3,  6000] loss: 0.251
[3,  8000] loss: 0.240
[3, 10000] loss: 0.231
[3, 12000] loss: 0.212
[3, 14000] loss: 0.205
Finished Training
```

In just three epochs the loss has decreased considerably.