

CSCI 315: Artificial Intelligence through Deep Learning

Prof. Levy

PyTorch Part 2: Learning

```

# Instantiate model, optimizer, and hyperparameter(s)
in_dim, feature_dim, out_dim = 784, 256, 10
lr=1e-3
loss_fn = nn.CrossEntropyLoss()
epochs=40
classifier = BaseClassifier(in_dim, feature_dim, out_dim)
optimizer = optim.SGD(classifier.parameters(), lr=lr)

def train(classifier=classifier,
          optimizer=optimizer,
          epochs=epochs,
          loss_fn=loss_fn):

    classifier.train()
    loss_lt = []
    for epoch in range(epochs):
        running_loss = 0.0
        for minibatch in train_loader:
            data, target = minibatch
            data = data.flatten(start_dim=1)
            out = classifier(data)
            computed_loss = loss_fn(out, target)
            computed_loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            # Keep track of sum of loss of each minibatch
            running_loss += computed_loss.item()
        loss_lt.append(running_loss/len(train_loader))
    print("Epoch: {} train loss: {}".format(epoch+1, running_loss/len(train_loader)))

```

```

# Instantiate model, optimizer, and hyperparameter(s)
in_dim, feature_dim, out_dim = 784, 256, 10
lr=1e-3
loss_fn = nn.CrossEntropyLoss()
epochs=40
classifier = BaseClassifier(in_dim, feature_dim, out_dim)
optimizer = optim.SGD(classifier.parameters(), lr=lr)

def train(classifier=classifier,
          optimizer=optimizer,
          epochs=epochs,
          loss_fn=loss_fn):

    classifier.train()
    loss_lt = []
    for epoch in range(epochs):
        running_loss = 0.0
        for minibatch in train_loader:
            data, target = minibatch
            data = data.flatten(start_dim=1)
            out = classifier(data)
            computed_loss = loss_fn(out, target)
            computed_loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            # Keep track of sum of loss of each minibatch
            running_loss += computed_loss.item()
        loss_lt.append(running_loss/len(train_loader))
    print("Epoch: {} train loss: {}".format(epoch+1, running_loss/len(train_loader)))

```

Cross-Entropy Loss Function

- Recall our simple error (loss) function from perceptrons and back-prop: $e_k = t_k - o_k$
- To compute the overall error (loss) value E for our classifier, we can just add up all the errors for each unit k :

$$E = \sum_k e_k$$

- Q: What is different about our softmax function for logistic regression?

Cross-Entropy Loss Function

- A: Softmax outputs a **probability distribution**.
A better loss function – *Cross-Entropy Loss*
– will take this into account.
- To understand Cross-Entropy, we should probably learn about **entropy** first.
- What do you think of when you think of entropy?

Cross-Entropy Loss Function

- A: Softmax outputs a probability distribution.
A better loss function – *Cross-Entropy Loss*
– will take this into account.
- To understand Cross-Entropy, we should probably learn about entropy first.
- What do you think of when you think of entropy?
- As with *convolution*, **entropy** has a more precise meaning (and positive connotation!) to engineers ...

Entropy: A Measure of Disorder

Entropy: A Measure of Disorder Information

- As is often the case, we use the same term (**entropy**) to refer to both a concept (**disorder**) and its apparent opposite (**information**).
- More precisely, entropy measures *the amount of information needed to characterize a system accurately*.



Much disorder: need to tell me position and rotation of each brick



Much order: just tell me number of rows and columns

Entropy: A Measure of Disorder Information

- Consider a simple coin toss: how much information is conveyed when I tell you what I got (heads or tails)?
- Intuitively, the probability of each toss is 0.5; i.e., I will guess correctly only 50% the time.
- However, as computer scientists, we would like to express information (entropy) as the **number of bits** needed. Putting it all together, we get ...

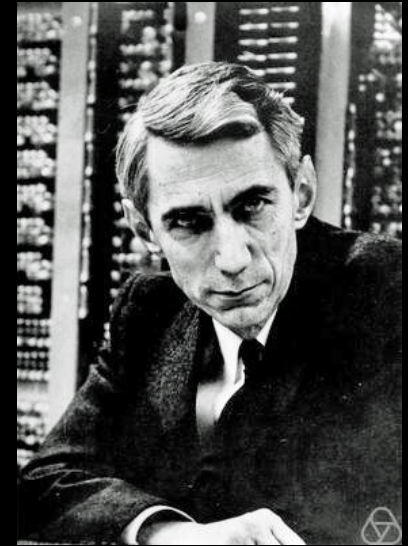
Entropy: A Measure of Disorder Information

$$H = - \sum_{i=1}^n p_i \log_2 p_i$$

Entropy: A Measure of Disorder Information

Entropy
(Greek H =
Roman E)

$$H = - \sum_{i=1}^n p_i \log_2 p_i$$



C. Shannon (1916-2001)

Entropy: A Measure of Disorder Information

Entropy
(Greek H =
Roman E)

$$H = - \sum_{i=1}^n p_i \log_2 p_i$$

- E.g., for a single coin toss:

Two possibilities (heads or tails)

$$H = - \sum_{i=1}^2 0.5 \log_2 0.5$$

```
>>> -np.sum([0.5 * np.log2(0.5)]*2)
1.0
```

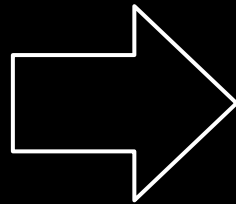
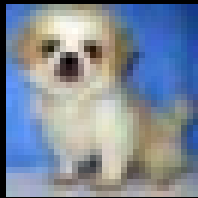
From Entropy to Cross-Entropy

- Once we know how to compute entropy, cross entropy is pretty simple: instead of one set of probabilities p , we have two (target value p and actual value q):

$$H(p, q) = - \sum_i p_i \log q_i$$

- Let's look at this for a hypothetical output from our dog/cat/mouse classifier

From Entropy to Cross-Entropy



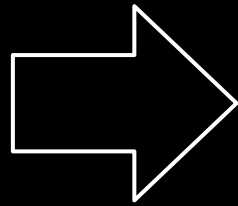
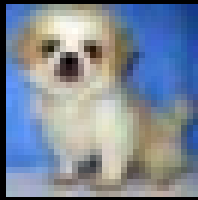
Classifier

q	p	
0.2	1	DOG
0.5	0	CAT
0.3	0	MOUSE

$$H(p, q) = - \sum_i p_i \log q_i$$

```
>>> q = [0.2, 0.5, 0.3]
>>> p = [1, 0, 0]
>>> -np.sum(p * np.log2(q))
2.321928094887362
```

From Entropy to Cross-Entropy



Classifier

q	p	
0.8	1	DOG
0.1	0	CAT
0.1	0	MOUSE

$$H(p, q) = - \sum_i p_i \log q_i$$

```
>>> q = [0.8, 0.1, 0.1]
>>> -np.sum(p * np.log2(q))
0.3219280948873623
```

Back to PyTorch!

```
# Instantiate model, optimizer, and hyperparameter(s)
in_dim, feature_dim, out_dim = 784, 256, 10
lr=1e-3
loss_fn = nn.CrossEntropyLoss()
epochs=40
classifier = BaseClassifier(in_dim, feature_dim, out_dim)
optimizer = optim.SGD(classifier.parameters(), lr=lr)

def train(classifier=classifier,
          optimizer=optimizer,
          epochs=epochs,
          loss_fn=loss_fn):

    classifier.train()
    loss_lt = []
    for epoch in range(epochs):
        running_loss = 0.0
        for minibatch in train_loader:
            data, target = minibatch
            data = data.flatten(start_dim=1)
            out = classifier(data)
            computed_loss = loss_fn(out, target)
            computed_loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            # Keep track of sum of loss of each minibatch
            running_loss += computed_loss.item()
        loss_lt.append(running_loss/len(train_loader))
    print("Epoch: {} train loss: {}".format(epoch+1, running_loss/len(train_loader)))
```

Stochastic Gradient Descent

In the algorithms we described in [“The Backpropagation Algorithm”](#), we used a version of gradient descent known as *batch gradient descent*. The idea behind batch gradient descent is that we use our entire dataset to compute the error surface and then follow the gradient to take the path of steepest descent. For a simple quadratic error surface, this works quite well. But in most cases, our error surface may be a lot more complicated. Let's consider the scenario in [Figure 4-6](#).

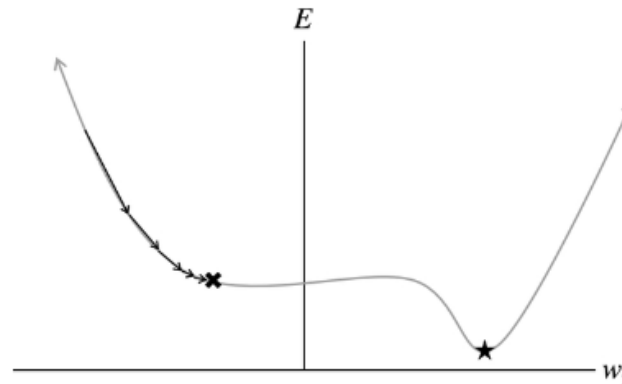


Figure 4-6. Batch gradient descent is sensitive to saddle points, which can lead to premature convergence

We have only a single weight, and we use random initialization and batch gradient descent to find its optimal setting. The error surface, however, has a flat region (also known as **saddle point** in

Stochastic Gradient Descent

Another potential approach is *stochastic gradient descent (SGD)*, where at each iteration, our error surface is estimated with respect to only a single example. This approach is illustrated by [Figure 4-7](#), where instead of a single static error surface, our error surface is dynamic. As a result, descending on this stochastic surface significantly improves our ability to navigate flat regions.

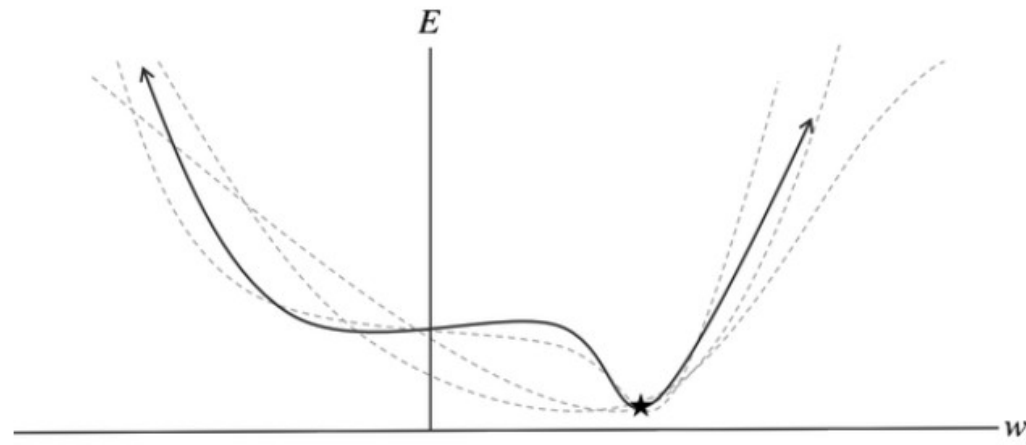


Figure 4-7. The stochastic error surface fluctuates with respect to the batch error surface, enabling saddle point avoidance

The major pitfall of SGD, however, is that looking at the error incurred one example at a time may not be a good enough approximation of the error surface. This, in turn, could potentially

Batch + Stochastic = Minibatch

make gradient descent take a significant amount of time. One way to combat this problem is using *minibatch gradient descent*. In minibatch gradient descent, at every iteration we compute the error surface with respect to some subset of the total dataset (instead of just a single example). This subset is called a *minibatch*, and in addition to the learning rate, minibatch size is another hyperparameter. Minibatches strike a balance between the efficiency of batch gradient descent and the local-minima avoidance afforded by stochastic gradient descent. In the context

```

# Instantiate model, optimizer, and hyperparameter(s)
in_dim, feature_dim, out_dim = 784, 256, 10
lr=1e-3
loss_fn = nn.CrossEntropyLoss()
epochs=40
classifier = BaseClassifier(in_dim, feature_dim, out_dim)
optimizer = optim.SGD(classifier.parameters(), lr=lr)

def train(classifier=classifier,
          optimizer=optimizer,
          epochs=epochs,
          loss_fn=loss_fn):

    classifier.train()
    loss_lt = []
    for epoch in range(epochs):
        running_loss = 0.0
        for minibatch in train_loader:
            data, target = minibatch
            data = data.flatten(start_dim=1)
            out = classifier(data)
            computed_loss = loss_fn(out, target)
            computed_loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            # Keep track of sum of loss of each minibatch
            running_loss += computed_loss.item()
        loss_lt.append(running_loss/len(train_loader))
    print("Epoch: {} train loss: {}".format(epoch+1, running_loss/len(train_loader)))

```

```

# Instantiate model, optimizer, and hyperparameter(s)
in_dim, feature_dim, out_dim = 784, 256, 10
lr=1e-3
loss_fn = nn.CrossEntropyLoss()
epochs=40
classifier = BaseClassifier(in_dim, feature_dim, out_dim)
optimizer = optim.SGD(classifier.parameters(), lr=lr)

def train(classifier=classifier,
          optimizer=optimizer,
          epochs=epochs,
          loss_fn=loss_fn):
    classifier.train()
    loss_lt = []
    for epoch in range(epochs):
        running_loss = 0.0
        for minibatch in train_loader:
            data, target = minibatch
            data = data.flatten(start_dim=1)
            out = classifier(data)
            computed_loss = loss_fn(out, target)
            computed_loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            # Keep track of sum of loss of each minibatch
            running_loss += computed_loss.item()
        loss_lt.append(running_loss/len(train_loader))
    print("Epoch: {} train loss: {}".format(epoch+1, running_loss/len(train_loader)))

```

▲
254

▼

`model.train()` tells your model that you are training the model. This helps inform layers such as Dropout and BatchNorm, which are designed to behave differently during training and evaluation. For instance, in training mode, BatchNorm updates a moving average on each new batch; whereas, for evaluation mode, these updates are frozen.



More details: `model.train()` sets the mode to train (see [source code](#)). You can call either `model.eval()` or `model.train(mode=False)` to tell that you are testing. It is somewhat intuitive to expect `train` function to train model but it does not do that. It just sets the mode.

```

# Instantiate model, optimizer, and hyperparameter(s)
in_dim, feature_dim, out_dim = 784, 256, 10
lr=1e-3
loss_fn = nn.CrossEntropyLoss()
epochs=40
classifier = BaseClassifier(in_dim, feature_dim, out_dim)
optimizer = optim.SGD(classifier.parameters(), lr=lr)

def train(classifier=classifier,
          optimizer=optimizer,
          epochs=epochs,
          loss_fn=loss_fn):

    classifier.train()
    loss_lt = []
    for epoch in range(epochs):
        running_loss = 0.0
        for minibatch in train_loader:
            data, target = minibatch
            data = data.flatten(start_dim=1)
            out = classifier(data)
            computed_loss = loss_fn(out, target)
            computed_loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            # Keep track of sum of loss of each minibatch
            running_loss += computed_loss.item()
        loss_lt.append(running_loss/len(train_loader))
    print("Epoch: {} train loss: {}".format(epoch+1, running_loss/len(train_loader)))

```

▲
433



In `PyTorch`, for every mini-batch during the *training* phase, we typically want to explicitly set the gradients to zero before starting to do backpropagation (i.e., updating the *Weights* and *biases*) because PyTorch *accumulates the gradients* on subsequent backward passes. This accumulating behaviour is convenient while training RNNs or when we want to compute the gradient of the loss summed over multiple *mini-batches*. So, the default action has been set to accumulate (i.e. sum) the gradients on every `loss.backward()` call.

Because of this, when you start your training loop, ideally you should zero out the gradients so that you do the parameter update correctly. Otherwise, the gradient would be a combination of the old gradient, which you have already used to update your model parameters, and the newly-computed gradient. It would therefore point in some other direction than the intended direction towards the *minimum* (or *maximum*, in case of maximization objectives).