CS 615 – Deep Learning

Learning

Slides adapted from material created by E. Alpaydin
Prof. Mordohai, Prof. Greenstadt, Pattern Classification (2nd Ed.),
Pattern Recognition and Machine Learning

Objectives

- Updating weights
- Online gradient descent
- Batch gradient descent

Updating Weights

- Ok fine, we can backpropagate the gradients.
- But how does this help us update the weights?
- When we hit a layer that has weights (for now, that's just the fully-connected layer) we can use the incoming (backcoming?) gradient to update the weights!
- For example, our fully-connected layers have weights, W, and biases, b
- So, we'll want to compute $\frac{\partial J}{\partial W}$ and $\frac{\partial J}{\partial b}$ and move/update our weights by going *some amount* in the direction of the gradient.

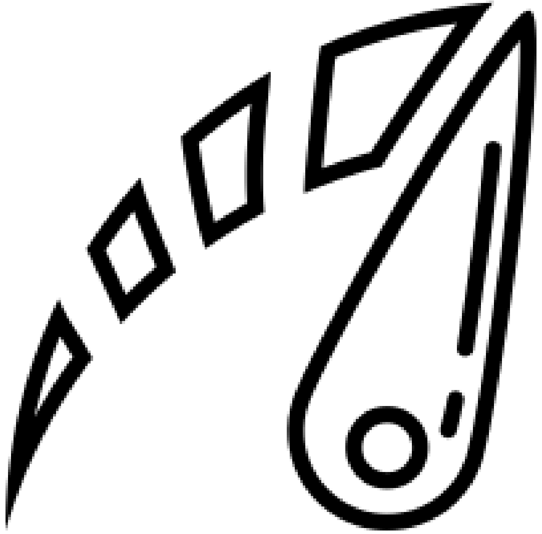
Gradient: Fully Connected Layer

- Imagine that our FC layer has a backpropagated gradient $\frac{\partial J}{\partial h} \in \mathbb{R}^{1\times K}$ coming into it.
- We then need $\frac{\partial h}{\partial W}$ and $\frac{\partial h}{\partial b}$ in order to compute $\frac{\partial J}{\partial W}$ and $\frac{\partial J}{\partial b}$
- Let's start with $\frac{\partial J}{\partial b}$
- What is the size of $\frac{\partial h}{\partial b}$?
  - Both are vectors, so this is a Jacobian matrix of size $\mathbb{R}^{K\times K}$
- What are the values in $\frac{\partial h}{\partial b}$?
  - Recall that $h = xW + b$
  - So $\frac{\partial h}{\partial b}$ is just an identity matrix!
- Chaining check?
  - $\frac{\partial J}{\partial b} = \frac{\partial J}{\partial h} \cdot \frac{\partial h}{\partial b} \in \left(\mathbb{R}^{1\times K}\right) \cdot \left(\mathbb{R}^{K\times K}\right) = \mathbb{R}^{1\times K}$
  - Same size as b!

Speed-Up

- Since $\frac{\partial h}{\partial b}$ is just an identity matrix, we can compute $\frac{\partial J}{\partial b}$ as

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial h}$$



Gradient: Fully Connected Layer

- How about $\frac{\partial J}{\partial W}$?
- What is the size of $\frac{\partial h}{\partial W}$?
  - h is a $1 \times K$ vector and $W \in \mathbb{R}^{D \times K}$
  - So $\frac{\partial h}{\partial W} \in \mathbb{R}^{K \times D \times K}$
  - We call a multidimensional matrix a **tensor.**

Gradient: Fully Connected Layer

- Before we get into the values of its elements, let's do our chaining check.
  - $\frac{\partial J}{\partial W} = \frac{\partial J}{\partial h} \cdot \frac{\partial h}{\partial W} \in \left(\mathbb{R}^{1\times K}\right) \cdot \left(\mathbb{R}^{K\times D\times K}\right) = \mathbb{R}^{1\times D\times K}$
  - This can be reduced to just being $\in R^{D\times K}$, which is the same size as W!
- So, what are the elements of $\frac{\partial h}{\partial W}$?

Gradient: Fully Connected Layer

$$h = xW + b$$

- What is $\frac{\partial h_k}{\partial W_{ij}}$?
  - If $k == j$

$$\frac{\partial h_k}{\partial W_{ij}} = x_i$$

  - Otherwise

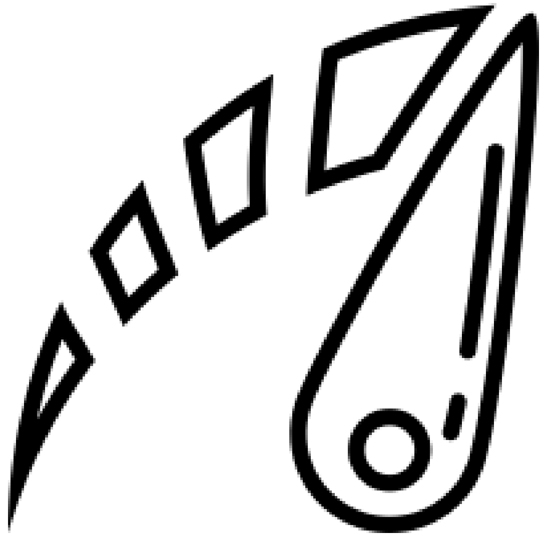
$$\frac{\partial h_k}{\partial W_{ij}} = 0$$

- How can we interpret this?
- The k^{th} matrix of the tensor will be all zeros except for the k^{th} column, which will have x^T on it!

Speed-Up

- Since each matrix is relatively sparce (only one column is populated) and the matrices are relatively redundant (the non-zero row of each matrix is identical), there are more efficient ways to compute $\frac{\partial J}{\partial W}$
- A common way is to *pre-multiply* the incoming gradient by x^T (the data coming into the FC layer, transposed).
- Sanity check…

$$\frac{\partial J}{\partial W} = x^T \frac{\partial J}{\partial h} \in \left(\mathbb{R}^{D\times 1}\right) \cdot \left(\mathbb{R}^{1\times K}\right) = \mathbb{R}^{D\times K}$$



Updating Weights

- Now that we have the gradient of our objective function with regards to its weights, we can update them!
- Since all our objective functions were framed as loss function, we actually want to go some amount in the direction **opposite** the gradient:

$$W^{(m)} = W^{(m)} + \eta\left(-\frac{\partial J}{\partial W^{(m)}}\right)$$

- The variable η is called the *learning rate* and it is a hyperparameter that we can choose.
- More on that in a moment, but as a start, we might set it to be a small value like $\eta = 10^{-4}$

Updating Weights

- Let's change our pseudocode a bit to include the update to the weights…

Let $\delta = \frac{\partial J}{\partial \hat{y}}$ //start gradient off as partial of objective function with regards to the output
Let $m = M$ //m is our current layer out of M total layers
while $m > 1$ //backprop until we hit the input layer
  if this layer has weights:
    Update its weights using the current value of δ and the partial of $h^{(m)}$ with regards to the weights.

  //update the gradient using the partial of the output of layer m, $h^{(m)}$, with regards to its input $h^{(m-1)}$
  $\delta = \delta \cdot \frac{\partial h^{(m)}}{\partial h^{(m-1)}}$

Forward-Backwards Propagation

- Now, for each layer we should have:
  - A forward method
  - A gradient method (returns a Jacobian matrix, or vector if using the speed-up)
  - A backwards method
    - Computes new gradient based on incoming gradient and the layer's gradient.
    - If the layer has weights, it updates them based on the incoming gradient.
- One training cycle (called an *epoch*) involves
  1. *forward propagating* from the input layer to the last layer.
  2. *back propagating* the gradient from the objective function back to the input layer, updating weights as you go.

Forward-Backwards Propagation

- Here's some real code!

```
L1 = layers.InputLayer(X)
L2 = layers.FullyConnected(X[1],classes.size)
L3 = layers.SoftmaxLayer()
L4 = layers.CrossEntropy()

layers = [L1, L2, L3, L4]

eta = 0.001

#forwards!
h = X
for i in range(len(layers)-1):
  h = layers[i].forward(h)

#backwards!
grad = layers[-1].gradient(Y,h)
for i in range(len(layers)-2,0,-1):
    grad = layers[i].backward(grad,eta)
```

Gradient Descent

- Do we just update the weights once?
  - No!
- We keep doing forward-backwards propagation until we hit some sort of convergence criteria
  - Or run out of time.
- What if we have more than one observation?
  - We typically do…

Online Gradient Descent

- One thing we could do is update the weights using each observation.
  - One iteration of doing this is referred to as an ***epoch****.*
- We call this *online* gradient descent since it can update weights as data comes in.
- Problem: Without taking into account all of the data at once (or at least some amount), it might take a long time to converge.

```
while still learning
     for each (x,y) in (X,Y)
          Perform forward propagation using x
          Perform backwards propagation using y, updating weights
     //end epoch
```

Batched Gradient Descent

- Problem: Without taking into account all of the data at once (or at least some amount), it might take a long time to converge.
- Solution: Accumulate the update rules across several observations and update the weights using their average.
- This is called *batched* gradient descent

```
while still learning
    (Re)set update accumulators to zero
    for each (x,y) in (X,Y)
        Perform forward propagation using x
        Perform backwards propagation using y, adding weight updates to accumulators
    Update weights using average of their accumulators
    //end epoch
```

Batched Gradient Descent

- Worth noting is that with batched gradient learning, the forward-backwards propagation of observations are independent of one another.
  - The weights don't get updated until all the observations computed their update rules.
- So, this is highly parallizeable
  - And can even be sped-up via linear algebra "tricks" (sparsity, etc..)

Batched Gradient Descent

- In fact, all the forward operations allow for processes a batch of inputs in the form of a matrix, X
- Let $X \in \mathbb{R}^{N \times D}$
- What is the dimension of $H = XW + b$?
- Backpropagating a batch just involves keeping the gradients per-observation, as tensors.
- What will have to change?

Batched Gradient Descent

- First off, our objective functions' gradients will now be $\in R^{N \times K}$
- Here's how this might look for the Least Squares objective function:

```
class LeastSquares():
  def eval(self,y,yhat):
    return (y - yhat).T @ (y - yhat)/y.shape[0]

  def gradient(self,y,yhat):
    return -2*(y-yhat)
```

Batched Gradient Descent

- Our layers' individual gradients will be $\in \mathbb{R}^{N \times K \times D}$ (or $N \times K$, if appropriate)
- Since our incoming gradient is $\in \mathbb{R}^{N \times K}$
  - If our layer's gradient is be $\in \mathbb{R}^{N \times K \times D}$ we'll need to multiply a matrix with a tensor.
  - If our layer's gradient is be $\in \mathbb{R}^{N \times K}$, we'll just Hadamard multiply the two matrices.
- Regardless, our outgoing gradient will be $\in \mathbb{R}^{N \times D}$
- While there are more efficient ways to do this, we could do the tensor product by just looping over the observations.
- Our abstract class's `backward` method may now look like:

```
def backward(self, gradIn, eta):
    sg = self.gradient()

    grad = np.zeros((gradIn.shape[0],sg.shape[2]))
    for n in range(gradIn.shape[0]):  #compute for each observation in batch
      grad[n,:] = gradIn[n,:]@sg[n,:,:]
    return grad
```

Batched Gradient Descent

$$\frac{\partial J}{\partial W} = x^T \frac{\partial J}{\partial h} \in \left(\mathbb{R}^{D\times N}\right)\cdot\left(\mathbb{R}^{N\times K}\right) = \mathbb{R}^{D\times K}$$

- How about updating our weights in a FC layer?
- For a batch, the gradient of the output of the FC layer with respect to W is now $\in \mathbb{R}^{N\times K\times D\times K}$
- Fortunately, we can still use our "trick" to compute $\frac{\partial J}{\partial W}$ using just the incoming gradient and the **transpose of the incoming data.**
- And now updating the biases of our FC layer can be done by just summing the input data over the observations!
- Here's how our FC layer's backward method might look:

```
def backward(self, gradIn, eta):
    gradOut = super().backward(gradIn,eta)

    pi = self.getPrevIn()
    po = self.getPrevOut()

    #get update for W
    dJdW = pi.T@gradIn

    #get update for b
    dJdb = np.sum(gradIn,0)

    #update weights
    self.__weights -= eta*dJdW/pi.shape[0]
    self.__biases -= eta*dJdb/pi.shape[0]

    return gradOut
```

Changes to our modules

- Let's make our layers support batching, since online gradient learning can be considered batching learning with a batch size of 1.
- What changes need to be made to our existing modules?
- Input Layer
  - No changes here!
- Objective Layers
  - Update the `eval` methods so that it is the *average* of the objective over all observations.

```
class LeastSquares():
  def eval(self,y,yhat):
    return (y - yhat).T @ (y - yhat)/y.shape[0]

  def gradient(self,y,yhat):
    return -2*(y-yhat)
```

Changes to our modules

- `forward` methods
  - No changed needed (assuming you implemented them correct).
- `gradient` method
  - Return the gradients tensors (or matrices).
- `backward` method
  - For FC layer update its weights and biases as mentioned.
  - Return updated gradient matrix using the incoming gradient matrix and module's gradient tensor (or matrix).

References

- Textbook
  - Section 4.3 : Gradient-Based Optimization
  - Section 5.9: Stochastic Gradient Descent
- Web
  - http://ruder.io/optimizing-gradient-descent/
  - https://www.mathworks.com/help/deeplearning/ref/connectlayers.html;jsessionid=5a96c52e8efec11d475011b1222f
  - https://deepnotes.io/softmax-crossentropy
  - https://ml-cheatsheet.readthedocs.io/en/latest/index.html