Problem Set 2

Nima Leclerc (nleclerc@seas.upenn.edu, PennID = nleclerc)

ESE 546 (Principles of Deep Learning)
School of Engineering and Applied Science
University of Pennsylvania
Total time = 12 hours

October 13, 2020

Problem 1:
(a) Note down all the peculiarities that you notice in the code. E.g., which one is better: using a batch-normalization layer before ReLU or after ReLU; what does this code do?
(b) What do the calls "model.train()" and "model.eval()" do and where do you use them in 9 a typical training and validation code? Why did we not have them in HW 1 when we wrote our own 10 library for training deep networks?
(c) Draw a rough picture of the resnet-18 architecture and note down the number of 12 parameters in each layer; you will find it easier to to write a function that computes the number of 13 parameters in each layer of the network.
(d) Weight decay should not be applied to the biases of the different layers in the network, 15 argue why this is the case.
(e) Write the code to iterate over all the network layers in resnet-18 and separate out the parameters in three groups: (i) batch-norm affine transform parameters, (ii) biases of convolutional and fully-connected layers, and (iii) all the rest. There is no need to submit the code separately as a Jupyter notebook in this case, since these are a few lines of code just copy them out into your PDF solutions.

Solution:

(a) One noticeable peculiarity in this implementation is that batch-normalization is applied prior to the ReLU, rather than after the ReLU. Batch-normalization is typically applied after the ReLU layer, since applying the ReLU prior to batchnorm will give a distribution of features to the following layer with non-zero mean and non-zero variance and the final output after the ReLU layer will have not been whitened correctly.

(b) "model.train()" will inform the model that it is in training mode. This allows layers like the batchnorm to switch to training mode during training. Likewise, "model.eval()" informs the model that it is in testing mode. This will now inform layers like batchnorm to switch testing mode during testing. In HW 1, the neural network was able to operate the same training and test mode (forward and backward modes) since we did not have layers like batch-norm involved. Batcnorm has an operation which computes the mean and variance over a minibatch. However, often in testing mode only one example is provided so the network will have to figure out a clever way to compute these statistical quantities without a minibatch (one alternative is on-the-fly mean/variance computation). During training mode, we would simply compute these means and variances from the minibatch of training examples themselves. Hence, two separate modes: "model.train()" and "model.eval()" will be needed.

(c) Figure depicts the resnet-18 architecture.

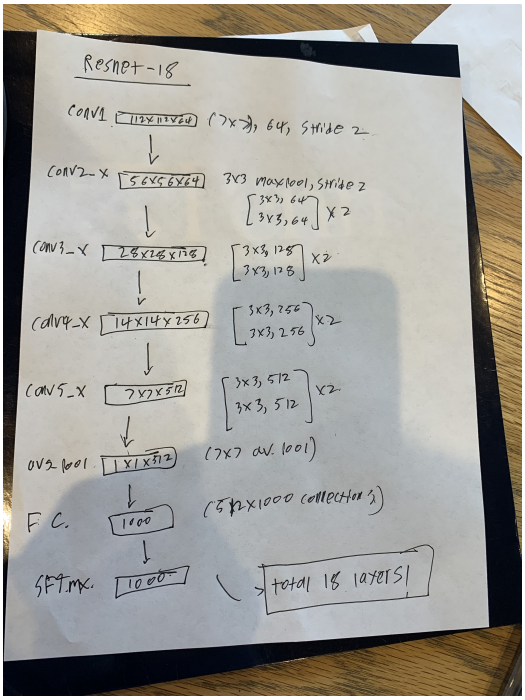

Figure 1: Resnet-18 architecture.

(d) In the case of a convolutional network like the one here, we should not apply weight decay to the biases in different layers since data coming into the model will have already been centered.

(e) Refer to the snippet of code below for an implementation of the separation of batch-norm affine transform parameters, biases of convolutional and fully-connected layer, and the rest.

```
resnet = resnet18()
bn_affine_weights = []
bn_affine_biases = []
conv_biases = []
fc_biases = []
other = []
for name, param in resnet.named_parameters():
    if 'bn' in name:
```

```
            if 'weight' in name:
                bn_affine_weights.append(param)
            elif 'bias' in name:
                bn_affine_biases.append(param)

        elif 'conv' in name:
            if 'bias' in name:
                conv_biases.append(param)

        elif 'fc' in name:
            if 'bias' in name:
                fc_biases.append(param)
        else:
            other.append(param)
print(bn_affine_weights)
print(bn_affine_biases)
print(conv_biases)
print(fc_biases)
print(other)
```

Problem 2:

Non-convex optimization problems are harder than convex optimization problems. There are however a few special non-convex problems that are easy. We will look at one of them here, namely unconstrained matrix factorization. Given a matrix $X \in \mathbb{R}^{m \times n}$ we would like to decompose it into two matrices of rank at most r

$$X = AB$$

where $A \in \mathbb{R}^{m \times r}$ and $B \in \mathbb{R}^{r \times n}$. Think of arranging all your data as columns of X. Columns of the matrix A are like elements of a dictionary, they correspond to different patterns in the data and are called atoms. The matrix B chooses which patterns to collect together in order to create a particular datum. Solving for factors A, B is usually done with constraint. We solve a simpler problem

$$A^\star, B^\star = argmin_{A,B} ||X - AB||_F^2$$

(a) Why is the above problem not convex?
(b) The global optimum for this loss function can be obtained easily in spite of it being non-convex; find it. You may find it useful to write down the SVD of X.

(c) Is the solution to the above optimization problem unique? Given one solution $A^\star, B^\star$ name one way using which you can obtain another solution.

Solution:

(a) We can show that the problem is not convex by first applying Jensen's equality. If Jensen's inequality holds, it imply that picking two points in the domain $(A_0, B_0), A_1, B_1)$ and drawing a line segment connecting the two on a function f, then f evaluated at the points defined within the domain of the line segment will be below the line segment. For $X \in \mathbb{R}^{m\times n}$. Take our function to be,

$$f(A, B) = ||X - AB||_F^2$$
$$= \sum_{i=1}^{m}\sum_{j=1}^{n} |X_{ij} - A_{ik}B_{kj}|^2$$

For now, let's just check the convexity with respect to on parameter and then generalize with respect to the other. Let's take A. For some probability θ such that $0 \leq \theta \leq 1$, we must have the following for a convex funciton

$$f(\theta A + (1-\theta)A', B) \leq \theta f(A, B) + (1-\theta) f(A', B)$$

$$f(\theta A + (1-\theta)A', B) = \sum_{i=1}^{m}\sum_{j=1}^{n} |X_{ij} - (\theta A_{ik} + (1-\theta)A'_{ik})B_{kj}|^2$$

$$= \sum_{i=1}^{m}\sum_{j=1}^{n} |X_{ij}|^2 + 2|X_{ij}(\theta A_{ik} + (1-\theta)A'_{ik})B_{kj}| + |(\theta A_{ik} + (1-\theta)A'_{ik})B_{kj}|^2$$

$$= \sum_{ij} |X_{ij}|^2 + 2(\theta|X_{ij}A_{ik}B_{kj}| + (1-\theta)|X_{ij}A'_{ik}B_{kj}|) + \theta^2|A_{ik}B_{kj}|^2 + (1-\theta)^2|A'_{ik}B_{kj}|^2$$

Combining terms corresponding to θ gives,

$$= \sum_{ij} |X_{ij}|^2 + \theta[|X_{ij}A_{ik}B_{kj}| + |A_{ik}B_{kj}|^2 + \theta|A_{ik}B_{kj}|^2] + (1-\theta)[|X_{ij}A'_{ik}B_{kj}| + (1-\theta)|A'_{ik}B_{kj}|^2]$$

$$= \sum_{ij} [\frac{1}{2}|X_{ij}|^2 + 2\theta|X_{ij}A_{ik}B_{kj}| + \theta^2|A_{ik}B_{kj}|^2] + [\frac{1}{2}|X_{ij}|^2 + 2(1-\theta)|X_{ij}A'_{ik}B_{kj}| + (1-\theta)^2|A'_{ik}B_{kj}|^2]$$

Comparing this with the expressions for $\theta f(A, B)$ and $(1-\theta)f(A', B)$.

$$\theta f(A, B) = \sum_{ij} [\theta |X_{ij}|^2 + 2\theta |X_{ij} A_{ik} B_{kj}| + \theta |A_{ik} B_{kj}|^2]$$

$$(1-\theta) f(A', B) = \sum_{ij} [(1-\theta) |X_{ij}|^2 + 2(1-\theta) |X_{ij} A'_{ik} B_{kj}| + (1-\theta) |A'_{ik} B_{kj}|^2]$$

Comparing these with the the two terms in the sum that we previously expanded shown below (first corresponding to θ and the second corresponding to $1-\theta$

$$\sum_{ij} [\frac{1}{2} |X_{ij}|^2 + 2\theta |X_{ij} A_{ik} B_{kj}| + \theta^2 |A_{ik} B_{kj}|^2]$$

$$\sum_{ij} [\frac{1}{2} |X_{ij}|^2 + 2(1-\theta) |X_{ij} A'_{ik} B_{kj}| + (1-\theta)^2 |A'_{ik} B_{kj}|^2]$$

It is clear that these expressions need not be strictly less than or equal to the corresponding expressions for $\theta f(A, B)$ and $(1-\theta)f(A', B)$ shown explicitly above. Geometrically speaking, different values of A in the domain can put f above or below the line segment. Hence,

$$f(\theta A + (1-\theta)A', B) \not\leq \theta f(A, B) + (1-\theta) f(A', B)$$

for all A in the domain. f is non-convex.

(b) We have,

$$f(A, B) = ||X - AB||_F^2$$

We want to solve,

$$\min_{A \in \mathbb{R}^{m \times r}, B \in \mathbb{R}^{r \times n}} ||X - AB||_F^2$$

We are unable to solve this problem using our standard methods because it is non-convex. We can write this as,

$$\min_{Y \in \mathbb{R}^{m \times n}} ||X - Y||_F^2$$

if we enforce that $\text{rank}(Y) = r$. This is a constraint that we're adding to the problem to make it solvable. Provided this form, we can you our standard means to solve for $A^\star$ and $B^\star$. We can take the SVD of Y ,

$$X \approx U \Sigma V^T$$

with $U \in \mathbb{R}^{m \times r}$, $\Sigma \in \mathbb{R}^{r \times r}$, and $V \in \mathbb{R}^{n \times r}$. U and V are orthonormal matrices and Σ is a diagonal matrix. Hence,

$$A^\star = U$$

$$B^\star = \Sigma V^T$$

(c) The solution to this optimization problem is not unique. A matrix and its inverse can be used to transform the factorized matrices for some positive permutation matrix R,

$$A^\star B^\star = A^\star R R^{-1} B^\star$$

From this, we get new matrices $A' = A^\star R$ and $B' = R^{-1} B^\star$. Hence,

$$A' = UR$$

$$B' = R^{-1} \Sigma V^T$$

for some arbitrary matrix positive permmuation matrix R. Since R can have multiple forms, the solution to the factorization *not unique*. Hence using the above relations, we can have multiple ways of obtaining a solution using R.

Problem 3:
Consider a loss function $l : \mathbb{R}^p \to \mathbb{R}$ that is invariant with respect to the scaling of the norm of its parameters $w \in \mathbb{R}^p$:

$$l(\lambda w) = l(w)$$

for any scalar $\lambda > 0$. (a) Show that for such a function, the gradient scales down by λ . We saw that is the case for the parameters of fully-connected layers (and thereby convolutional ones as well) with batch-normalization. You will find it useful to write down the definition of the derivative.
Due to this scale invariance, weight decay cannot affect the parameters of these layers at all. However, in practice, networks with batch-normalization have a much better validation error with weight-decay than without. This seems contradictory: if weight decay does not change the loss function in the presence of BN then how can it obtain a better validation error? We will try to understand this next.

(b) Consider the gradient descent update

$$w^{t+1} = w^t - \eta \nabla l(w^t)$$

and show that if we define the direction of the weights as

$$v^t = \frac{w^t}{||w^t||_2}$$

the update in (1) can be written as

$$v^{t+1} = v^t - \frac{\eta}{||w||_2^2}(I - v^t(v^t)^T)\nabla l(v^t) + \mathcal{O}(\eta^2)$$

The coefficient

$$\eta' = \frac{\eta}{||w^t||_2^2}$$

is the effective learning rate. The multiplier $I - v^t(v^t)^T$ is a matrix that transforms the gradient $\nabla l(v^t)$ linearly, we can think of this matrix as a rotation and scaling of the gradient.
Observe now that the weights may grow unbounded because batch-normalization does not differentiate between weights of different magnitudes. If this happens, the effective learning rate η' decreases drastically which essentially freezes the direction of the weights v^t. Weight decay counteracts this effect, it encourages the magnitude of the weights to go down which keeps the effective learning rate large and allows the direction of the weights to change. You will do well to remember that in the presence of batch-normalization, only the direction of the weights in a deep network matters, the magnitude of the weights does not matter.

Solution:

(a) If we have a loss function l such that $l(\lambda w) = l(w)$, we can show the derivative would come out to $(\frac{dl(\lambda w)}{d(\lambda w)})$. Let's take the definition of a derivative to expand out this derivative,

$$\frac{dl(\lambda w)}{d(\lambda w)} = \lim_{\Delta w \to 0} \frac{l(\lambda w + \lambda \Delta w) - l(\lambda \Delta w)}{\lambda \Delta w}$$

$$= \cdot \frac{1}{\lambda} \lim_{\Delta w \to 0} \frac{l(\lambda(w + \Delta w))) - l(\lambda \Delta w)}{\Delta w}$$

$$= \frac{1}{\lambda} \lim_{\Delta w \to 0} \frac{l(w + \Delta w) - l(\Delta w)}{\Delta w}$$
$$= \frac{1}{\lambda} \frac{dl(w)}{dw}$$

Hence,

$$\frac{dl(\lambda w)}{d(\lambda w)} = \frac{1}{\lambda} \frac{dl(w)}{dw}$$

(b) If we define the normalized weights as $v^t = \frac{w^t}{||w^t||_2}$, substituting this into our standard gradient update step gives

$$||w^{t+1}||_2 v^{t+1} = ||w^t||_2 v^t - \eta \nabla_{w^t} l(||w^t||_2 v^t)$$

$$v^{t+1} = \frac{1}{||w^{t+1}||_2}(||w^t||_2 v^t - \eta \nabla_{||w^t||_2 v^t} l(||w^t||_2 v^t))$$
$$= \frac{1}{||w^{t+1}||_2}[v^t - \frac{\eta}{||w^t||_2^2} \nabla_{v^t} l(v^t)]$$

We can approximate,

$$\frac{1}{||w^{t+1}||_2} \approx \frac{1}{||w^t||_2 + \mathcal{O}(\eta)}$$

Performing a Taylor expansion on this gives,

$$\frac{1}{||w^t||_2 + \mathcal{O}(\eta)} \approx -\frac{1}{\mathcal{O}(\eta)}(1 + ||w^t||_2 + \mathcal{O}(\eta^2))$$

$$\approx -(1 + ||w^t||_2 + \mathcal{O}(\eta^2))$$

Now expand this out for our update step,

$$v^{t+1} \approx -(1 + ||w^t||_2 + \mathcal{O}(\eta^2))(v^t - \frac{\eta}{||w^t||_2^2} \nabla_{v^t} l(v^t))$$

$$\approx -(1 + ||w^t||_2)(v^t - \frac{\eta}{||w^t||_2^2} \nabla_{v^t} l(v^t)) + \mathcal{O}(\eta^2)$$

$$\approx v^t(1 - ||w^t||_2) - \frac{\eta \nabla_{v^t} l(v^t)}{||w^t||_2}(1 + \frac{1}{||w^t||_2}) + \mathcal{O}(\eta^2)$$

$$= v^t - \frac{\eta \nabla_{v^t} l(v^t)}{||w^t||_2} (\frac{1 + \frac{1}{||w^t||_2}}{1 - ||w^t||_2}) + \mathcal{O}(\eta^2)$$

$$\approx v^t - \frac{\eta \nabla_{v^t} l(v^t)}{||w^t||_2^2} (I - \frac{1}{||w^t||_2^2} w^t (w^t)^T) + \mathcal{O}(\eta^2)$$

$$= v^t - \frac{\eta \nabla_{v^t} l(v^t)}{||w^t||_2^2} (I - v^t (v^t)^T) + \mathcal{O}(\eta^2)$$

Hence, we have

$$v^{t+1} = v^t - \frac{\eta}{||w||_2^2} (I - v^t (v^t)^T) \nabla l(v^t) + \mathcal{O}(\eta^2)$$

Problem 5:
Please refer to attached Jupyter notebook for this problem.

Problem 6:
Training of recurrent neural networks (RNNs) is often difficult because of the vanishing or exploding gradient problem. We will study this in a simple setting without any nonlinearities.
(a) If the input to an RNN at the t^{th} timestep is $x^t \in \mathbb{R}^d$ and the hidden vector is $z^t \in \mathbb{R}^p$, the hidden vector z^{t+1} is given by

$$z^{t+1} = \sigma(w_x x^t + w_z z^t)$$

where $w_x \in \mathbb{R}^{p \times d}$ and $w_z \in \mathbb{R}^{p \times p}$ are weights and σ is a nonlinearity. If $\sigma(z) = z$, and $w_x = 0$, then we have

$$z^{t+1} = w_z z^t$$

Write down the back-propagation gradient for w_z if the loss function is only a function of the hidden vector at time T. Compute the conditions on the weight matrix w_z under which the gradient explodes and vanishes.
(b) Argue how the nonlinearity σ affects the exploding/vanishing gradients. Which nonlinearities are well-suited for training RNNs?
(c) Updates to the weights are computed using SGD as

$$w_z^{new} = w_z^{new} - \eta \frac{dl}{dw_z}$$

We would like to protect the weights w_z^{new} from blowing up even if the gradient $\bar{w}_z$ explodes. Can you think of a way to do this? Similarly, can you modify these updates to handle the vanishing gradient problem?
(d) Explain how an LSTM solves the problem of vanishing gradients.

Solution:

(a) Provided that the loss function has the form,

$$J^{(t+1)}(h^{(t)}) = J^{(t+1)}(w_z h^t)$$

Then the backprop gradient is given by,

$$\frac{J^{(i)}}{\partial h^{(j)}} = \frac{J^{(i)}}{\partial h^{(i)}} \prod_{j<t\leq i} \frac{\partial h^{(t)}}{\partial h^{(t-1)}}$$

With $h^{(t)} = W_z h^{(t-1)}$,

$$= \frac{J^{(i)}}{\partial h^{(i)}} W_z^{(i-j)} \prod_{j<t\leq i} \frac{\partial h^{(t)}}{\partial h^{(t-1)}}$$

Taking the norm of both sides,

$$||\frac{J^{(i)}}{\partial h^{(j)}}||_2 \leq ||\frac{J^{(i)}}{\partial h^{(i)}}||_2 ||W_z||_2^{(i-j)} \prod_{j<t\leq i} ||\frac{\partial h^{(t)}}{\partial h^{(t-1)}}||_2$$

We see that if $||W_z||_2 < 1$, the gradient will decrease exponentially (vanish) with $(i-j)$. However, if $||W_z||_2 > 1$, the gradient will increase exponentially (explode).

(b) A linear activation as shown in this problem will give exploding/vanishing effects during training. This is because the linear activation function is unbounded. Hence, we instead want a function that is bounded. tanh and sigmoidal nonlinearities will saturate at large W. These functions are bounded. Hence, we can avoid the exploding problem by using a bounded function like tanh or sigmoid function.

(c) One thing we can do to prevent the weights from being effected by exploding gradients in the update step is to scale the gradient down if the norm of

the gradient is greater than some threshold value. This can be done at each iteration of training, prior to feeding the gradient into the gradient update step. Hence, we would take a step in the same intended direction but the step would be smaller. One way of scaling down would be to normalize by the norm of the 'exploding' gradient. We can handle the problem of vanishing gradients a similar way by instead adding a very small bias to the gradients after each iteration to prevent them from dropping down to zero.

(d) LSTM solves the problem of vanishing gradients by preserving information about the gradients over multiple timesteps. It remembers everything on a timestep for a particlular cell, and then it preserves that information indefinetley. It's more difficult for the RNN example we used earlier to learn the W_z weight matrix that prefers the hidden state.