ESE 546, FALL 2020
HOMEWORK 2
INSTRUCTOR SOLUTIONS

Solution 1. Understanding the Resnet 18 architecture

(a) The code uses ReLU after the batch-norm. This is because having a batch-norm before the activation function ensures that the inputs to the activation functions are normalized. This would imply that the outputs of the activations are unlikely to be saturated (happens often for sigmoid and tanh). However, the other order will also be a reasonable choice since this would imply that the input to any layer would be of the same scale. Furthermore, batch norm before ReLU would imply that any bias term from the convolution layer has no effect. See the Lecture Notes for a more detailed discussion.

(b) Batch-norm and dropout operate differently during the train and test phases. For batch-norm, the normalizing mean and variance during test time is not the batch-mean, but rather the running average over multiple batches. Similarly, dropout scales the activations by $1 - p$, instead of sampling which activations to zero out (like in train time).

HW1 does not use these two operations and hence operates in an identical fashion, in both training and testing regimes. However, other architectures have to switch between these two modes, and hence require model.train() and model.eval().

(c) A diagram of resnet18 is present in Figure 1. The number of parameters in each block is

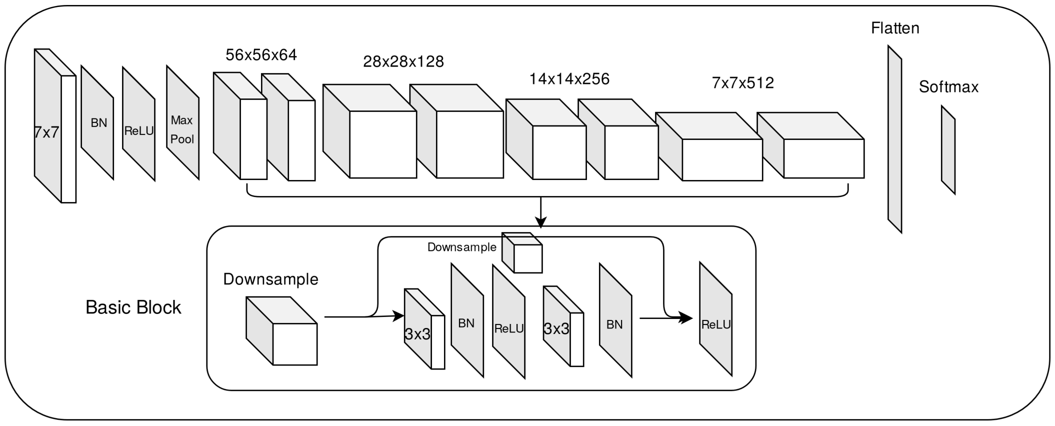


FIGURE 1. ResNet-18

presented in Table 1. The number of parameters in each of the 18 layers are: (conv1':

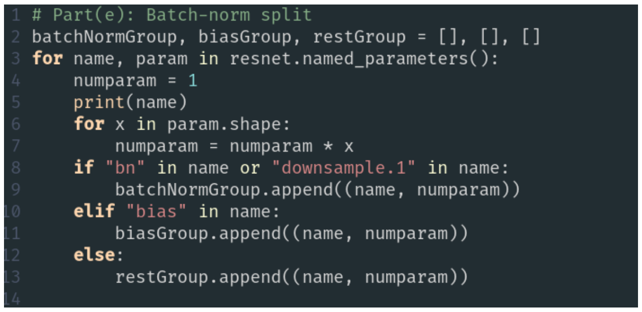
7x7	Basic Block 1	Basic Block 2	Basic Block 3	Basic Block 4	FC	Total
9408	147968	525568	2099712	8393688	513000	11698512

TABLE 1. Parameters in each block

9536) (layer1-0-conv1: 36992), (layer1-0-conv2: 36992), (layer1-1-conv1: 36992), (layer1-1-conv2: 36992), (layer2-0-conv1: 73984), (layer2-0-conv2: 147712), (layer2-0-downsample: 8448), (layer2-1-conv1: 147712), (layer2-1-conv2: 147712), (layer3-0-conv1: 295424), (layer3-0-conv2: 590336), (layer3-0-downsample: 33280), (layer3-1-conv1: 590336), (layer3-1-conv2: 590336), (layer4-0-conv1: 1180672), (layer4-0-conv2: 2360320), (layer4-0-downsample: 132096), (layer4-1-conv1: 2360320), (layer4-1-conv2: 236032), and (fc: 513000).

(d) Under linear transformations of the input $f(x) = x - b$, the corresponding weights for the transformed inputs would have have the bias increased by b. However, the weight decay will penalize this increase. This is common for domains like images where all pixels are of the same scale, but the right bias is not known. In order to make our neural network invariant to these transforms, we do not penalize the bias terms.

(e) Code for seperating out bias parameters



```python
# Part(e): Batch-norm split
batchNormGroup, biasGroup, restGroup = [], [], []
for name, param in resnet.named_parameters():
    numparam = 1
    print(name)
    for x in param.shape:
        numparam = numparam * x
    if "bn" in name or "downsample.1" in name:
        batchNormGroup.append((name, numparam))
    elif "bias" in name:
        biasGroup.append((name, numparam))
    else:
        restGroup.append((name, numparam))
```

The batch-norm, biases and the other parameters are indicated in the table.

BatchNorm	9600
Biases params	1000
Rest params	11678912
Total params	11689512

Solution 2. (a) The multiplication of A and B produces terms that are of the form $\Sigma(a_{ij}b_{jk})$ which is a bilinear optimization problem and hence non-convex.

(b) We will use the following result. If σ_i are the singular values of a matrix $C \in \mathbb{R}^{m\times n}$

$$\sum_i \sigma_i^2(C) = \|C\|_F^2 = \sum_{i,j} C_{ij}.$$

This is because if the singular vectors of C are $v_1, \dots, v_r$, they span all rows of C and therefore for any row c_j of the matrix C we have

$$\sum_{i=1}^{r} \langle c_j, v_i \rangle = \|c_j\|_2^2.$$

The above identity can now be obtained by summing up over all rows $j \leq m$.

The problem asks us to minimize the Frobenius norm of $C := X - AB$ so we need to make sure C has as small singular values as possible. Let the SVD of X be

$$X = U \Sigma V^\top.$$

If we set

$$A^* = U_r \sqrt{\Sigma_r}$$
$$B^* = \sqrt{\Sigma_r} V_r^\top$$

where U_r are the first r columns of U, Σ_r are the top r singular values of X and V_r are the first r columns of V, then we have

$$\|X - A^* B^*\|_F^2 = \sum_{i=r+1}^{\text{rank}(X)} \sigma_i(X)^2$$

which is the minimal value of the problem.

(c) The values of A^* and B^* obtained above are not unique. For instance we can multiply A by any constant $e \in \mathbb{R}$ and B by $1/e$ to preserve the objective. In general, any

$$A = A^* Q$$
$$B = Q^\top B^*$$

where Q is an orthogonal matrix gives a different solution. We can also have Q to be a permutation matrix which shuffles the columns of A^* and rows of B^* correspondingly to preserve the objective. This shows that even if the global minimum of the optimization problem that we have found above has the same value, there are multiple (an infinite number in fact) solutions that give the same value.

This problem is an example of a linear auto-encoder, and in general, neural networks are full of such symmetries (you are encouraged to think about this for a two-layer network with ReLU nonlinearities).

Solution 3. (a) Because many students think that the relationship should be opposite, let's look at the following example:

$$f(x, y) = \frac{y}{x}$$

which is clearly a scale invariant function and calculate its gradient: $\nabla f(x,y) = \begin{bmatrix} -\frac{y}{x^2} \\ \frac{1}{x} \end{bmatrix}$
Now if we evaluate it at $\lambda(x,y)$, the relationship is clear. We can think the notation $\nabla_{\lambda x} f(\lambda x)$ in 2 ways. Firstly, as calculating the gradient of $f(x)$ and then evaluating that gradient at the point λx like we did above. Secondly, (more pedantic, but avoids mistakes in most cases) using the definition of the partial derivatives, i.e. applying a perturbation h at the point λx ($\lambda \neq 0$).

$$\frac{\partial f(\lambda x)}{\partial \lambda x_i} = \lim_{h\to 0} \frac{f(\lambda \mathbf{x}_{-i}, \lambda x_i + h) - f(\lambda \mathbf{x}_{-i}, \lambda x_i)}{h} \tag{1}$$

To exploit the scale invariance, we will change the variable in the limit and set $\alpha = h/\lambda$. It is clear that when $h \to 0$, also $\alpha \to 0$. So, (3) is equivalent to:

$$= \lim_{\alpha\to 0} \frac{f(\lambda \mathbf{x}_{-i}, \lambda x_i + \lambda\alpha) - f(\lambda \mathbf{x}_{-i}, \lambda x_i)}{\lambda\alpha} = \frac{1}{\lambda} \lim_{\alpha\to 0} \frac{f(\lambda(\mathbf{x}_{-i}, x_i + \alpha)) - f(\lambda(\mathbf{x}_{-i}, x_i))}{\alpha} \tag{2}$$

from scale invariance of f,

$$= \frac{1}{\lambda} \lim_{\alpha\to 0} \frac{f((\mathbf{x}_{-i}, x_i + \alpha)) - f((\mathbf{x}_{-i}, x_i))}{\alpha} = \frac{1}{\lambda} \frac{\partial f(x)}{\partial x_i}$$

(b) dividing the equation by $\|x^{(k+1)}\|$ we get:

$$u^{(k+1)} = \frac{x^k - \eta \nabla f(x^k)}{\|x^k - \eta \nabla f(x^k)\|} \tag{3}$$

Looking at the denominator, we sense that since η is small, this is not far away from the norm of our current position. We take the Maclaurin expansion of the right hand side w.r.t η, to quantify this intuition. We denote $g(\eta) = x^k - \eta \nabla f(x^k)$ for convenience. g is a function of η and we will omit it when it is clear. So the right hand side can be written as $h(\eta) = g(\eta)/\|g(\eta)\| = h(0) + h'(0)\eta + \mathcal{O}(\eta^2)$.

$$h(0) = \frac{x^k}{\|x^k\|} = u^k$$

And by chain rule ($\cdot$ is the standard matrix-vector multiplication),

$$h'(0) = (\nabla_g \frac{g}{\|g\|}) \cdot g'(\eta)\Big|_{\eta=0} = (\frac{1}{\|g(\eta)\|}(I - \frac{g(\eta)g(\eta)^T}{\|g(\eta)\|^2})) \cdot (-\nabla f(x^k))\Big|_{\eta=0}$$

$$= (\frac{1}{\|x^k\|}(I - \frac{x^k {x^k}^T}{\|x^k\|^2})) \cdot (-\frac{1}{\|x^k\|} \nabla f(u^k))$$

where we changed the variable x^k to its scaled version u^k using the scale invariance property of our function. Hence,

$$u^{(k+1)} = h(\eta) = u^k - \frac{\eta}{\|x^k\|^2}(I - u^k {u^k}^T)\nabla f(u^k) + \mathcal{O}(\eta^2)$$

Remark you can alternatively consider the right hand side of (3) as a function of x and expand it around x^k.

Remark $P = I - uu^T$ is a projection matrix in the direction orthogonal to u

Remark Many students claimed that since $\lambda = 1/\|x^k\|$ and thus a function, we cannot apply a). However, the scale invariant property holds for any x and for any lambda (even if you calculated your lambda afterwards, based on your x). For a fixed x, $\lambda(x)$ is also fixed, and when we calculate the gradients, we *first* take the gradient and *then* evaluate it at the point $\lambda(x)x$. Convince yourselves by substituting a $\lambda = \lambda(x, y)$ in the example above.

Remark (Something from the Office hours) Mind the difference between $f'(g(x))$ and $f(g(x))' = f'(g(x))g'(x)$. Staying in 1-D for simplicity, if $u = u(x)$ then writing: $\frac{\partial f(x)}{\partial x}|_{x*} = \frac{\partial f(u)}{\partial u}|_{u(x*)} \cdot \frac{\partial u(x)}{\partial x}|_{x*}$ is still an abuse of notation that causes many mistakes, since the term: $\frac{\partial f(u)}{\partial u}|_{u(x*)}$ is actually $\frac{\partial (f \circ u^{-1})(y)}{\partial y}|_{u(x*)}$, the reparametrized version of f. That is why, changing the variable inside the gradient, using chain rule and (not scale invariance) as follows:

$$\nabla f(x^k) = J_x(u)\nabla f(u^k) = \frac{1}{\|x^k\|}(I - u^k {u^k}^T)\nabla f(u^k)$$

and then using the fact $P = (I - u^k {u^k}^T)^2 = I - u^k {u^k}^T$ may appear to give the same result but it is not what we are looking for, since the right-most gradient, is the reparametrized version of f and not the gradient of f evaluated at u^k.

Solution 4. Moved to the next homework.

Solution 5. See the Jupyter notebook.

Solution 6. (a) We will use the notation $\overline{w_z}$ shown in the Lecture Notes. The forward propagation computes

$$z^{t+1} = (w_z)^t \, z_0.$$

The backward propagation will therefore be

$$\begin{aligned}\frac{\mathrm{d}\ell}{\mathrm{d}w_z} &= \overline{w_z} \\ &= \overline{z^{t+1}} \, \frac{\mathrm{d}z^{t+1}}{\mathrm{d}w_z} \\ &= \overline{z^{t+1}} \, \left(t \, w_z^{t-1}\right) \, z_0.\end{aligned}$$

Consider $t+1 = T$. We do not need to worry about $\overline{z^T} = \frac{\mathrm{d}\ell(z^T)}{\mathrm{d}z^T}$ for the argument here. The norm of the gradient on weights w_z is

$$\|\overline{w_z}\|_2 = \|\overline{z^{t+1}}\|_2 \, t \|w_z^{t-1} z_0\|_2.$$

Let us assume that weights $A := w_z \in \mathbb{R}^{p \times p}$ are diagonalizable, i.e., we can write $A = T\Lambda T^{-1}$ and get

$$A^k z_0 = T\Lambda^k T^{-1} z_0 = \sum_{i=1}^{p} \alpha_i \lambda_i^k v_i$$

where λ_i are the eigenvalues of A, v_i are the corresponding eigenvectors and α_i are the rows of the matrix $T^{-1} z_0$. This shows that if the inner product between α_i and v_i corresponding to an eigenvalue with magnitude greater than 1 is non-zero, the norm

$$\|A^k \, z_0\|_2 \to \infty.$$

On the other hand if the largest magnitude of eigenvalues of A is smaller than 1, or if the vector z_0 is such that the $\alpha_i v_i = 0$ for all eigenvalues $|\lambda_i| \geq 1$ then the norm

$$\|A^k \, z_0\|_2 \to 0.$$

This example demonstrates that depending upon the initial value of the hidden vector in an RNN, and the weights, we may get exploding or vanishing gradients for the weights.

(b) The recursive forward pass with saturating nonlinearities like tanh and sigmoid is problematic because the activations do not change much even if the inputs to the nonlinearity change a lot; this is particularly problematic for long sequence lengths and these nonlinearities therefor often result in vanishing gradients. ReLU can result in vanishing or exploding gradients, the former happens if the inputs to ReLU are negative. Exploding gradients are more common with ReLUs because the input magnitude is unchanged by the nonlinearity and the therefore the eigenvalues of the weight matrices in the fully-connected parts of the RNN (as also for standard deep networks) determine the magnitude of the output. Overall, all nonlinearities can cause problems with exploding/vanishing gradients in RNNs.

(c) Exploding/vanishing gradients are problematic if we update the weights

$$w^{t+1} = w^t - \eta \, \frac{\mathrm{d}\ell}{\mathrm{d}w}$$

using SGD. If gradient magnitude is large, since the gradient on a mini-batch is only a stochastic estimate of the true gradient over the entire dataset, we are updating the weights by a large value in potentially the wrong direction. If the gradient magnitude is very small, the weights are essentially frozen and training is stalled.

A common solution to this problem is to modify the gradient $\frac{\mathrm{d}\ell}{\mathrm{d}w_z}$ before updating the weights. For instance, we can scale the gradient to have a fixed ℓ_2 norm c, clip various entries of the gradient vector to lie between $[-c, c]$ for some constant c.

(d) See lecture notes for more detail. An LSTM network has three gates that update and control the cell states, which are the forget gate, input gate and output gate. The forget gate and the input gate contain learned parameters and gives the training more control over gradient vanishing. A heuristic argument of how the backprop gradient does not have multiplicative terms of the weights in an LSTM is enough for full credit.

Observe that the gradient through $T - t$ time steps of an RNN is

$$\frac{\partial z^T}{\partial z^t} = \prod_{i=1}^{T-t} w_z \sigma'(w_z z^{T-i}),$$

where z is the hidden state and w_z is the weight. We ignored the inputs x and biases b in the gradient. Similarly, the gradient through $T - t$ time steps of an LSTM is

$$\frac{\partial c^T}{\partial c^t} = \prod_{i=1}^{T-t} f^{t+i} + \text{other terms},$$

where c is the cell state (memory of the cell in the Lecture notes) and $f^{t+1} = \sigma(w_{hf} h^t + w_{xf}\ x^{t+1})$ is the forget gate. We ignore the other terms in the gradient because they are not essential to the argument in this question. The gradient decays with $w_z \sigma(\cdot)$ for an RNN while the gradient decays with $\sigma(\cdot)$ for an LSTM. Hence an LSTM is able to tackle the problem of exploding/vanishing much better than an RNN.