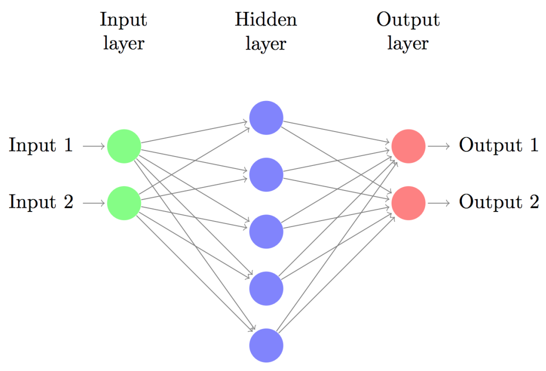

Figure 1: Neural Network

0.1 Problem

Compute the gradients and derivatives of the loss function of the given neural network as shown in figure 1 with respect to the parameters: W_1, W_2, b_1 and b_2. Where W_1, W_2 are the weight matrices; and b_1 and b_2 are bias the vectors. Let $x \in \mathcal{R}^2$, $W_1 \in \mathcal{R}^{2*500}$, $b_1 \in \mathcal{R}^{500}$, $W_2 \in \mathcal{R}^{500*2}$ and $b_2 \in \mathcal{R}^2$. Also, show how the forward and backpropagation algorithms work.

Note: This question serves as good exercise to see how forward propagation works and then how the gradients are computed to implement the backpropagation algorithm. Also, the reader will get comfortable with the computation of vector, tensor derivatives and vector/matrix calculus. A useful document can be found at (https://compsci697l.github.io/docs/vecDerivs.pdf) for the interested reader to get familiar with tensor operations.

0.2 Solution

Let us first compute the forward propagation. Let x be the input. The first hidden layer is computed as follows:

$$z_1 = xW_1 + b_1 \tag{1}$$

We then apply a non-linear activation function to equation 1

$$a_1 = tanh(z_1) \tag{2}$$

The output layers' activation are obtained using the following transformation

$$z_2 = a_1 W_2 + b_2 \tag{3}$$

Finally, a *softmax* is applied to equation 2 to get:

$$a_2 = \hat{y} = softmax(z_2) \tag{4}$$

where $\hat{y}$ is the predicted output by the feedforward network

Let's see this feedforward network through a circuit diagram as illustrated in figure 2. Now, let's see how the derivatives are computed with respect to the hidden nodes and bias vectors. Refer figure 3.

We have to compute the derivative of the loss function with respect to W_1, W_2, b_1 and b_2 that is see the effect of these parameters on the loss function (which we actually have to minimize).

Below are the steps to compute the various gradients as shown in figure 3:

- $$Loss = y * ln(\sigma(z_2)) + (1 - y) * ln(1 - \sigma(z_2)) \tag{5}$$

  Also, note that:

  $$\frac{d\sigma(z_2)}{dz} = ln(\sigma(z_2)) * ln(1 - \sigma(z_2)) \tag{6}$$

  Therefore,

  $$\frac{\partial Loss}{\partial z_2} = y - \sigma(z_2) = y - \hat{y} \tag{7}$$

  Since, $\sigma(z_2)$ = $\hat{y}$

- $$\frac{\partial z_2}{\partial w_2} = \frac{\partial(a_1 W_2 + b_2)}{\partial w_2} = a_1 \tag{8}$$

- $$\frac{\partial z_2}{\partial b_2} = 1 \tag{9}$$

- $$\frac{\partial z_2}{\partial tanh(z_1)} = \frac{\partial(a_1 W_2 + b_2)}{\partial tanh(z_1)} = \frac{\partial(tanh(z_1) W_2 + b_2)}{\partial tanh(z_1)} = z_2 \tag{10}$$

- $$\frac{\partial tanh(z_1)}{\partial z_1} = 1 - tanh^2(z_1) \tag{11}$$

- $$\frac{\partial tanh(z_1)}{\partial z_1} = 1 - tanh^2(z_1) \tag{12}$$

- $$\frac{\partial z_1}{\partial W_1} = \frac{\partial(x W_1 + b_1)}{\partial W_1} = x \tag{13}$$

- $$\frac{\partial z_1}{\partial b_1} = \frac{\partial(x W_1 + b_1)}{\partial b_1} = 1 \tag{14}$$

Finally, we can now use the chain rule to compute the effect of the four parameters namely W_1, W_2, b_1 and b_2 on the Loss function.

In what follows, $(P)^T$ indicates the transpose of some matrix or vector P.

- $$\frac{\partial Loss}{\partial W_2} = \frac{\partial Loss}{\partial z_2} * \frac{\partial z_2}{\partial W_2} = a_1^T(y - \hat{y}) \tag{15}$$

- $$\frac{\partial Loss}{\partial b_2} = \frac{\partial Loss}{\partial z_2} * \frac{\partial z_2}{\partial b_2} = (y - \hat{y}) \tag{16}$$

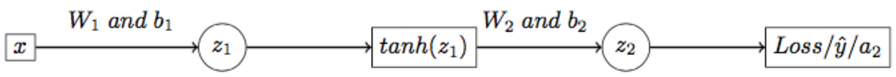

Figure 2: Feedforward Circuit Diagram

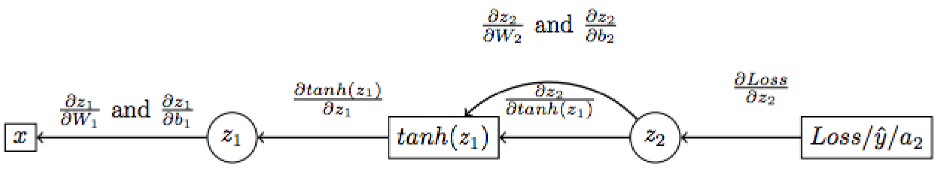
$\frac{\partial z_2}{\partial W_2}$ and $\frac{\partial z_2}{\partial b_2}$
$\frac{\partial z_1}{\partial W_1}$ and $\frac{\partial z_1}{\partial b_1}$
$\frac{\partial tanh(z_1)}{\partial z_1}$
$\frac{\partial z_2}{\partial tanh(z_1)}$
$\frac{\partial Loss}{\partial z_2}$
x
z_1
$tanh(z_1)$
z_2
$Loss/\hat{y}/a_2$

Figure 3: Backpropagation Circuit Diagram

-
$$\begin{aligned}\frac{\partial Loss}{\partial W_1} &= \frac{\partial Loss}{\partial z_2} * \frac{\partial z_2}{\partial z_1} * \frac{\partial z_1}{\partial W_1} &(17)\\ &= \frac{\partial Loss}{\partial z_2} * \frac{\partial z_2}{\partial tanh(z_1)} * \frac{\partial tanh(z_1)}{\partial z_1} * \frac{\partial z_1}{\partial W_1} &(18)\\ &= x^T(1 - tanh^2 Z_1)(y - \hat{y})W_2^T &(19)\end{aligned}$$

-
$$\begin{aligned}\frac{\partial Loss}{\partial b_1} &= \frac{\partial Loss}{\partial z_2} * \frac{\partial z_2}{\partial z_1} * \frac{\partial z_1}{\partial W_1} &(20)\\ &= \frac{\partial Loss}{\partial z_2} * \frac{\partial z_2}{\partial tanh(z_1)} * \frac{\partial tanh(z_1)}{\partial z_1} * \frac{\partial z_1}{\partial b_1} &(21)\\ &= (1 - tanh^2 Z_1)(y - \hat{y})W_2^T &(22)\end{aligned}$$

Hence, we have computed both the forward propagation and back propagation for the given multi-layer neural network.

1 Machine Learning: Gradient Checking

1.1 Problem

It is a good practice to use gradient checking to compare the analytic and numerical gradient. Oftentimes, we see the absolute error i.e. during the iterations of the gradient descent algorithm the absolute value of the difference between the consecutive gradients are checked. The algorithm terminates when the absolute error is less than some small threshold.
But, using absolute error is not recommended and rather the use of relative error is suggested. Prove this statement with a numerical example.

1.2 Solution

Let g_n and g_{n+1} be the two consecutive numerical gradients.

$|g_n$ - $g_{n+1}|$ is the absolute error which is calculated and if the error is less than some threshold then the gradient check is successful.

If g_n and g_{n+1} are both very close to 1. For example g_n= 1 and g_{n+1} = 1.0001 then
$|g_n$ - $g_{n+1}|$ is of the order 10^{-4}
and the gradient check looks good.

But, when g_n and g_{n+1} both have very low values in the order of 10^{-6}; then an absolute error of 10^{-4} will be considered very large and hence gradient check will fail.

Therefore, it is better to use the relative error defined as $\frac{|g_n - g_{n+1}|}{|g_n + g_{n+1}|}$

This is the ratio between the absolute difference to the absolute sum of the two gradients. This method thus scales the error and gives us the correct error estimate than when only the absolute error is used.

UPDATE: Instability can be encountered when performing the gradient check. Here, I ask one more question which seems interesting.
Question
Kink(s) can appear in some activation functions. Give an example of one such activation function. Comment on one precaution which you might want to take care of when doing the gradient check on such an activation function.
Answer
ReLU is one such example. When there is kink(s), one needs to monitor the values of *xh* and *x+h*. If these two values are on two different sides of a kink, one should exclude this gradient check.

2 Function Representation and Network Capacity

2.1 Problem

Let us say that we are given two types of activation functions: linear and a hard threshold function as stated below:
Linear:

$$y = w_0 + \sum_i w_i x_i \tag{23}$$

Hard Threshold:

$$y = \begin{cases} 1, & \text{if } w_0 + \sum_i w_i x_i \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{24}$$

Which of the following can be exactly represented by a neural network with ***one hidden layer***? You can use linear and/or threshold activation functions. Justify your answer with a brief explanation.

1. polynomials of degree 2

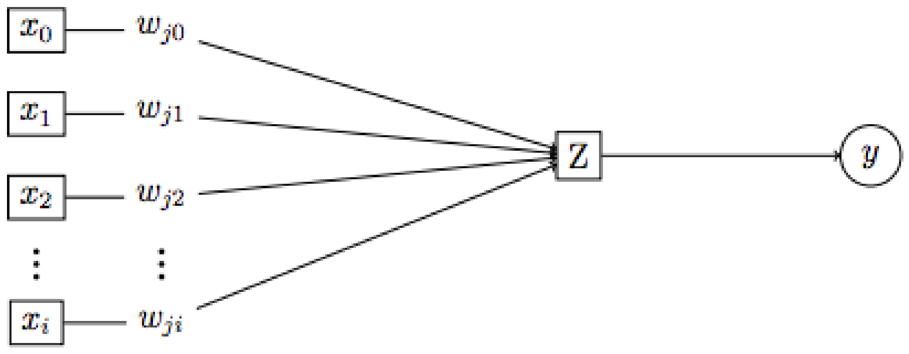

Figure 4: Figure for Question 3

2. polynomials of degree 1
3. hinge loss
4. piecewise constant functions

2.2 Solution

1. Polynomials of degree 2: **NO**
   Let's say that there is a polynomial of degree 2 such as,

   $$y = ax^2 + bx + c \tag{25}$$

   We cannot get the first term of degree two (x^2) as a linear combination of x and 1. There is no way to get a squared term with the given specification of the neural network.

2. Polynomials of degree 1: **YES**

   Consider the neural network as given in Figure 4.

   It is of the form

   $$y = wx + b \tag{26}$$

   This is a linear combination of the inputs and weights which are of degree one and can be represented by the linear activation function.

3. Hinge loss: **NO**
   With ***one hidden layer*** the hinge loss cannot be represented as it will require a combination of both of the given activation functions viz. linear and threshold.

   But, we can combine the linear and threshold activation functions in a two hidden layer network to get the hinge loss.

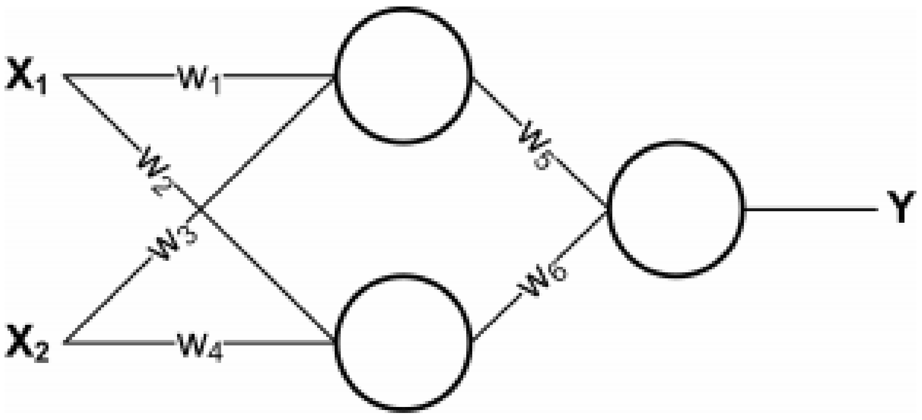

Figure 5: Figure for Question 4

4. Piecewise constant functions: **YES**

   We can represent a piece-wise function in the 1D case where each input is hard-thresholded to obtain

$$x \geq \frac{-w_0}{w_j} \tag{27}$$

3 Activation Functions

3.1 Problem

Consider a neural network as shown in Figure 5. The network has linear activation functions. Let the various weights be defined as shown in the figure and also the output of each unit is multiplied by some constant *k*. Answer the following questions.

1. Re-design the neural network to compute the same function without using any hidden units. Express the new weights in terms of the old weights. Draw the obtained perceptron.
2. Can the space of functions that is represented by the above artificial neural network also be represented by linear regression?
3. Is it always possible to express a neural network made up of only linear units without a hidden layer? Give a brief justification.
4. Let the hidden units use sigmoid activation functions and let the output unit use a threshold activation function. Find weights which cause this network to compute the XOR of X_1 and X_2 for binary-valued X_1 and X_2. Assume that there are no bias terms.

3.2 Solution

1. Connect the input X_1 to an output unit with weight (w_5 w_1 + w_6 w_2) and connect the input for X_2 to the output unit with weight (w_5 w_3 + w_6 w_4).

   See Figure 5 for the perceptron. k^2 is the activation function. where,

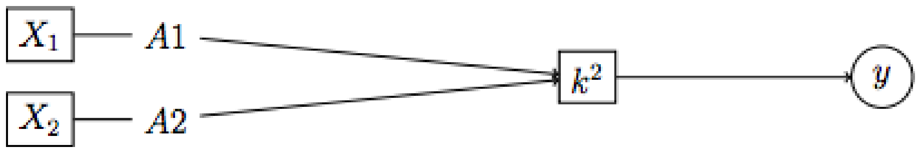

Figure 6: Figure for Solution 4(part:1)

$A_1 = w_5\ w_1 + w_6\ w_2$ and
$A_2 = w_5\ w_3 + w_6\ w_4$

2. Yes, any function in the above network has the form:

   $Y = \alpha_1 + \alpha_2$
   $Y = k^2\ (w_5\ w_1 + w_6\ w_2)\ X_1 + k^2\ (w_5\ w_3 + w_6\ w_4)\ X_2$

   This can be thought of as a linear regression on the inputs X_1 and X_2 with coefficients α_1 and α_2

3. True. Each layer can be thought of as performing a matrix multiplication to find its representation given the representation on the layer that it receives input from. Thus the entire network just performs a chain of matrix multiplications. Therefore we can simply multiply all the matrices together to find the weights to represent the function with a single layer.

4. One solution: $w_1 = w_3 = 10$,$w_2 = w_4 = 1$,$w_5 = 5$, and $w_6 = 6$. The intuition here is that we can decompose A XOR B into (A OR B) AND NOT (A AND B). We make the upper hidden unit behave like an OR by making it saturate when either of the input units are 1. It is not possible to make a hidden unit that behaves exactly like AND, but we can at least make the lower hidden unit continue to increase in activation after the upper one has saturated.

4 Time complexity of Linear and Quadratic Models

4.1 Problem

This is a three part problem.

1. Comment on the computational complexity of the least square method for Linear Regression. Give a step by step account of various matrix computations. Also, compare the least square method with the gradient descent method. Given that there are N training examples and *k* features.

2. Further assume that you have two datasets which has the same features. The two datasets are labeled as 1 and 2. Given that you have already computed $X_1^T X_1$, $X_1^T y_1$ and $X_2^T X_2$, $X_2^T y_2$. Then, what is the time complexity of the training the combined dataset?

Suppose we want to learn a quadratic model:

$$\begin{array}{llllllll} y = & w_0 & + & w_1x_1 & + & w_2x_2 & + & w_3x_3 & + & & \dots & w_kx_k & + \\ & & & w_{11}x_1^2 & + & w_{12}x_1x_2 & + & w_{13}x_1x_3 & + & & \dots & w_{1k}x_1x_k & + \\ & & & & & w_{22}x_2^2 & + & w_{23}x_2x_3 & + & & \dots & w_{2k}x_2x_k & + \\ & & & \vdots & & & & \vdots & & & & \vdots & \\ & & & & & & & & & w_{k-1,k-1}x_{k-1}^2 & + & w_{k-1,k}x_{k-1}x_k & + \\ & & & & & & & & & & + & w_{k,k}x_k^2 & \end{array}$$

Figure 7: Figure for Question 5 (part:3)

3. Consider the model in Figure 7. We would like that quadratic model. Comment on the computational complexity of the model. You can use explanation from part 1 of the question. Answer the following:

   - In big-O notation what would be the computational complexity of learning the MLE weights using matrix inversion?
   - What would be the computational complexity of one iteration of the batch gradient descent method?

4.2 Solution

1. Consider the following least square equation where symbols have the usual meaning:

$$\theta = (X^TX)^{-1}X^Ty \tag{28}$$

   We have the following computations:

   - $O(k^2$N) to multiply X^T by X
   - $O(k$N) to multiply X^T by y
   - $O(k^3)$ to compute the inverse and the final multiplication.
   - Asymptotically speaking $O(k^2$N) dominates $O(k$N), so we can ignore the $O(k$N) part. Also, since we have assumed that the matrix is non-singular, it implies that $O(k^2$N) asymptotically dominates $O(k^3)$.

     Therefore, the total time complexity is $O(k^2$N).
     Note: We can play around with various methods of matrix decompositions such LU or QR factorization to get different time complexities as required for the task in hand.

   - There is more to Gradient Descent and Stochastic Gradient Descent. For each iteration: Gradient Descent takes N time units for the N summations where each of the gradient computation takes $O(k)$. Therefore, Batch Gradient Descent takes $O($N$k)$. Stochastic Gradient Descent takes 1 time unit. A good reference material can be found here: http://leon.bottou.org/publications/pdf/compstat-2010.pdf

2. The time complexity for such a formulation is $O(k^3)$ since we just add the relevant matrices and vectors and then solve for kxk linear system.

3. There k^2 weights and N examples. On similar reasoning from part 1,

   - $O(Nk^4 + k^6)$ where the first term is for the matrix multiplication of $X^T X$ and the second term is for the inversion or asymptotically speaking $O(Nk^4)$
   - $O(Nk^2)$

5 Optimization, Transfer Learning and Feature Interpretation

5.1 Problem

Answer the following questions. Give an intuitive example and/or a mathematical expression where possible.

1. Give some scenarios of Transfer Learning where it can be used in a ConvNet

2. Compare and contrast the following hyperparameter optimization techniques:

   - Grid Search
   - Random Search
   - Bayesian Optimization
   - Gradient-based Optimization

3. Explain and give a mathematical expression for each of the following three concepts with respect to a ConvNet:

   - Equivalence
   - Equivariant
   - Invariant

5.2 Solution

1. Transfer Learning can be used in the following scenarios in a CNN:

   - We can remove the last fully connected layer from a ConvNet and then use the rest of the network. This network can be used as a feature extractor for some other task. This is especially useful when the task at hand has similar data as the network used for pre-training. This greatly removes the overhead of computing features again and again.
   - One can also use a pre-trained network to fine tune the weights of the current ConvNet. Also, one can fix the earlier layers and only fine-tune the weights of the upper layers.

2. Hyperparameter optimization or better known as model selection is the problem of choosing a set of hyperparamters for a learning algorithm.

- ***Grid Search*** also known as parameter sweeping is an exhaustive searching through a manually specified subset of the hyperparameter space of the learning algorithm. Since the parameter space might include real-valued or unbounded value spaces for certain parameters, manually set bounds and discretization may be necessary before applying grid search. Let us say that there are two hyperparamters that needs to be optimized α and β.

  $\alpha \in \{0.1, 0.01, 0.001\}$
  $\beta \in \{1, 2, 3, 4\}$

  We will have to swap over 3^4 = 12 possibilities. Clearly, this method is computationally expensive.
- ***Random Search*** This simply samples parameter settings a fixed number of times. Random search has found to be more effective in high-dimensional spaces than exhaustive (grid) search. This is because often times, it turns out some hyperparameters do not significantly affect the loss. Therefore having randomly dispersed data gives more **textured** data than an exhaustive search over parameters that ultimately do not affect the loss.
- ***Bayesian Optimization*** Bayesian optimization consists of developing a statistical model of the function from hyperparameter values to the objective evaluated on a validation set. Intuitively, the methodology assumes that there is some smooth but noisy function that acts as a mapping from hyperparameters to the objective. Bayesian optimization relies on assuming a very general prior over functions which when combined with observed hyperparameter values and corresponding outputs yields a distribution over functions. This method is found to be the best as it trade-offs the exploration and exploitation of the functions.
- ***Gradient based optimization*** One can also also compute gradients over the hyperparameters using automatic differentiation and gradient descent. Though, this technique is seldom used.

3. The three representations are described as follows:

   Image representations such as HoG, SIFT, SURF and CNNs can be thought of as functions ϕ mapping an image x ∈ X to a vector $\phi(x) \in R^d$

   - ***Equivariance*** A representation ϕ is equivariant with a transformation *g* of the input image if the transformation can be transferred to the representation output. Formally, equivariance with *g* is obtained when there exists a map M_g: $R^d \to R^d$

     For every x:
     $\phi(gx)$ = $M_g\phi(x)$
     A sufficient condition for the existence of M_g is that the representation ϕ is invertible.

     Example: Any convolutional representation or Dense SIFT is equivariant to translations of the input image as this result in a translation of the feature field. Rotations in HoG features is another example of the equivariant represntation.

- ***Invariant*** Invariance is a special case of equivariance obtained when M_g (or a subset of M_g) acts as the simplest possible transformation, i.e. the identity map. Invariance is often regarded as a key property of representations since one of the goals of computer vision is to establish invariant properties of images.

  Example: the category of the objects contained in an image is invariant to viewpoint changes. A CNN is made for this purpose: pooling layer!

- ***Equivalence*** While equi/invariance look at how a representation is affected by transformations of the image, equivalence studies the relationship between different representations.

  k = $\phi_1 \to \phi_2$
  ϕ_1 = E_k ϕ_2

  A good reference is: https://arxiv.org/pdf/1411.5908.pdf

6 Normalization Techniques

6.1 Problem

Batch normalization normalize the activations over the current batch in each hidden layer, generally right before the non-linearity. Given the following neural network which has a Batch Normalization layer, compute the flow of gradients through the Batch Norm layer. Show the steps of your calculation.

Consider for a input batch *x* of size (N,D) which goes through a hidden layer of size *H* which has some weights *w* of size *(D,H)* and a bias **b** of size *H*. The following is then a possible neural network representation:

1. Affine Transformation

$$h = XW + b \tag{29}$$

   the size of H is (N,H)

2. Batch Normalization Transformation

$$y = \gamma\hat{h} + \beta \tag{30}$$

$$\hat{h}_{kl} = (h_{kl} - \mu_l)(\sigma_l^2 + \epsilon)^{-1/2} \tag{31}$$

   where

$$\mu_l = (1/N)\sum_k (h_{pl}) \tag{32}$$

   and

$$\sigma_l^2 = (1/N)\sum_k (h_{pl} - \mu_l)^2 \tag{33}$$

with k=1,.....,N and l=1,...,H

3. Also, we can add a non-linearity such as ReLu:

$$a = ReLu(y) \tag{34}$$

Note that γ and β are learnable parameters.

6.2 Solution

We need to calculate the following gradients:

$\frac{\partial L}{\partial \gamma}$, $\frac{\partial L}{\partial \beta}$ and $\frac{\partial L}{\partial h}$

By the chain rule we know that,

$$\frac{\partial L}{\partial h_{ij}} = \sum_{k,l} \frac{\partial L}{\partial y_{kl}} \frac{\partial y_{kl}}{\partial \hat{h}_{kl}} \frac{\partial \hat{h_{kl}}}{\partial h_{ij}} \tag{35}$$

Let's keep the notation simplified and instead of considering the row by row and column by column of the involved Hessian, let us only consider the derivatives of the functions.

Since, an explicit loss function is not given, let's assume the loss function to ***L***. Therefore, $\frac{\partial \hat{L}}{\partial y}$ can be left as this expression.

1. $\frac{\partial \hat{h}}{\partial h}$

$$\frac{\partial \hat{h}}{\partial h} = \frac{\partial (h-\mu)(\sigma^2+\epsilon)^{-1/2}}{\partial h} = (\sigma^2+\epsilon)^{-1/2} - (1/2)(h-\mu)(\sigma^2+\epsilon)^{-3/2}\frac{d\sigma^2}{dh} \tag{36}$$

where,

$$\frac{d\sigma^2}{dh} = \frac{d(1/N)(h-\mu)^2}{dh} = (2/N)(h-\mu) \tag{37}$$

Therefore,

$$\frac{\partial \hat{h}}{\partial h} = \frac{\partial (h-\mu)(\sigma^2+\epsilon)^{-1/2}}{\partial h} = (\sigma^2+\epsilon)^{-1/2} - (1/N)(h-\mu)^2(\sigma^2+\epsilon)^{-3/2} \tag{38}$$

Hence,

$$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial y}(\gamma)(\frac{\partial (h-\mu)(\sigma^2+\epsilon)^{-1/2}}{\partial h} = (\sigma^2+\epsilon)^{-1/2} - (1/N)(h-\mu)^2(\sigma^2+\epsilon)^{-3/2}) \tag{39}$$

2.

$$\frac{\partial y}{\partial \hat{h}} = \frac{\partial(\gamma\hat{h}+\beta)}{\partial \hat{h}} = \gamma \tag{40}$$

3.

$$\frac{dL}{d\beta} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \beta} = \frac{\partial L}{\partial y} \tag{41}$$

$$\frac{dL}{d\gamma} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \gamma} = \frac{\partial L}{\partial y}(\hat{h}) \tag{42}$$

We have successfully computed the error gradients for a BatchNorm Neural Network.

7 Question 1

This question asks the student to derive the forward and backward equations for a 1 layer MLP with a tanh activation. It's a solid, well written question and the derivation appears to be correct. The diagrams are also nice and it is well polished.

Score: 3/3

8 Question 2

This question asks the user about computing gradients numerically to test whether one has implemented backpropagtion correctly. The main idea of the answer is that one should use relative error instead of absolute error for this, but since the real/estimated gradients ought to be identical, I'm not sure how interesting this is. I'm also not sure if numerical instability is usually that big of a problem when checking a backpropagation implementation.

I like the idea of asking about numerical differentiation, but I would like to see more substance and variety in the answer. For example, how about a demonstration using a package like theano or pytorch?

Score: 2.5/3

9 Question 3

Nice question about how 1-layer MLPs with a threshold activation can learn different functions.
Score: 3/3

10 Question 4

This questions asks the student to explain how deep neural networks with identity activation collapse to a linear function. It also asks about how xor can be computed using a neural network.
Score: 3/3

11 Question 5

This question asks the student to compare gradient descent for learning linear regression against the closed-form solution for ordinary least squares, with a focus on computation complexity.
Score: 2.5/3

12 Question 6

This question gives nice comparisons of different ways of doing hyperparameter tuning, an explanation of transfer learning in convnets, as well as an explanation of how different types of learned and fixed features (like SIFT) capture invariance or equivariance.
Score: 3/3

13 Question 7

This questions asks the student to compute the gradients through a layer with batch normalization. This is a good exercise and seems clear enough to read.
Score: 2.5/3