ECE 595: Homework 1

Rahul Deshmukh, PUID: 0030004932

(Fall 2019)

1 Problem 1: XOR Function

$$\mathcal{X} = \{0,1\} \times \{0,1\} \quad f^* : \mathcal{X} \to \{0,1\}$$

$$f^*(\boldsymbol{x}) = \begin{cases} 1, & \text{if } x_1 \neq x_2 \\ 0, & \text{otherwise} \end{cases}$$

(a) No, we cannot learn XOR function f^* using a linear function in $\mathbb{R}^2$. We can clearly see from figure 1 that we cannot draw a straight line in x_1-x_2 plane such that the data points with label as 1 and 0 are on two different sides of the line. The best classification we can do in this plane is with a 50% accuracy.

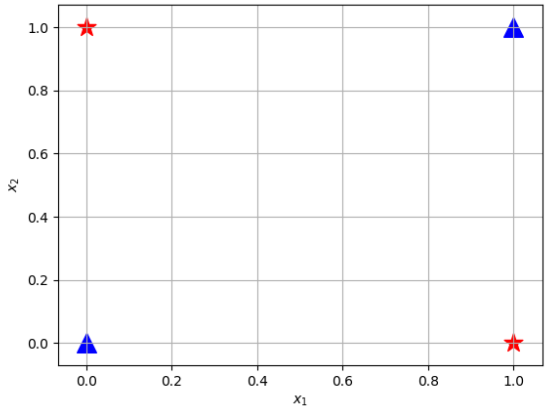


Figure 1: XOR function, with star for $f^* = 1$ and triangle for $f^* = 0$

(b) A feed-forward with non-linear activation function can learn the XOR function f^*. With the help of a hidden layer with **ReLU** activation can help us in learning the f^*. The feed-forward net f is given by:

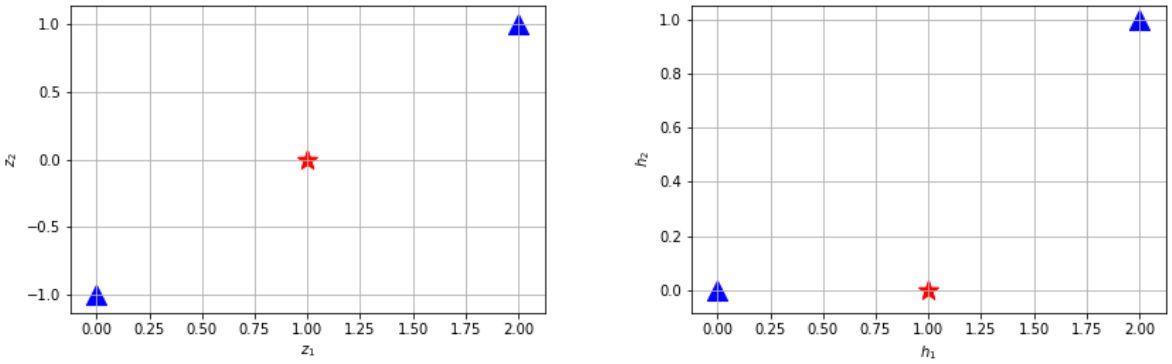
$$f(\boldsymbol{x}) = \boldsymbol{w}^T \,\mathrm{ReLU}(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c}) + b \tag{1}$$

$$\text{where} \quad \mathrm{ReLU}(z_i) = \max\{0, z_i\}$$

$$\text{with :} \quad \boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \quad b = 0$$

The first transformation $\boldsymbol{Z} = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c}$ is a linear transformation. This moves the points in figure 1 to as shown in figure 2(a). It can be seem from figure 2(a) that the data points are not linearly separable.



(a) Output of Linear Transformation (**Z**)

(b) Output of Hidden Layer (**H**)

Figure 2: data points with star for $f^* = 1$ and triangle for $f^* = 0$

The second transformation is using the **ReLU**() function as the hidden layer and the output of the hidden layer $\boldsymbol{H} = \mathrm{ReLU}(\boldsymbol{Z})$ can be seen in figure 2(b). The data points are now linearly separable and hence we can now classify them using a linear classifier.

Thus with the feed forward f as defined in Eq 1 we can learn the XOR function f^* correctly.

2 Problem 2: RBF kernel

The kernel function is defined as:

$$K : \mathbb{R}^d \times \mathbb{R}^d, \quad (\boldsymbol{x}, \boldsymbol{x'}) \to \exp\{-\gamma \|\boldsymbol{x} - \boldsymbol{x'}\|_2^2\}$$

The RBF kernel corresponds to a dot product in an infinite dimensional vector space even when d=1. The proof is as follows:

Let u,v $\in \mathbb{R}^1$ and $\gamma = 1$

$$
\begin{aligned}
K(u,v) &= \exp\{-\|u-v\|_2^2\} \\
&= \exp\{-(u-v)^2\} \\
&= \exp\{-u^2\}\exp\{2uv\}\exp\{-v^2\} \\
&\text{Using Taylor expansion for } \exp\{2uv\} \\
&= \exp\{-u^2\}\Big(\sum_{k=0}^{\infty}\frac{2^k u^k v^k}{k!}\Big)\exp\{-v^2\} \\
&= \exp\{-u^2\}\Big(1, \sqrt{\frac{2^1}{1!}}u^1, \sqrt{\frac{2^2}{2!}}u^2, \sqrt{\frac{2^3}{3!}}u^3, ...\Big)^T\Big(1, \sqrt{\frac{2^1}{1!}}v^1, \sqrt{\frac{2^2}{2!}}v^2, \sqrt{\frac{2^3}{3!}}v^3, ...\Big)\exp\{-v^2\} \\
&= \boldsymbol{\phi}^T(u)\boldsymbol{\phi}(v) \\
&\text{where} \quad \boldsymbol{\phi}(x) = \exp\{-x^2\}\Big(1, \sqrt{\frac{2^1}{1!}}x^1, \sqrt{\frac{2^2}{2!}}x^2, \sqrt{\frac{2^3}{3!}}x^3, ...\Big)
\end{aligned}
$$

Thus, the RBF kernel is a dot product of a infinite dimensional vector space, which means its VC dimension is infinity. It can fit to any training data but cannot generalize well for testing data.

3 Problem 3: Gradient -Based Learning

(a) In machine learning we are always trying to find a solution $\boldsymbol{\theta^*}$ which minimizes a loss function $\mathcal{L}(\boldsymbol{\theta})$. The loss function for neural networks is usually composed of several layers which have non-linear activation which usually results in a non-convex loss function. In order to find the solution we make use of gradient of the loss function, $(\nabla_{\boldsymbol{\theta}}\mathcal{L})$ and move in the opposite direction of the gradient with a user-defined learning rate (α). This is called gradient-based learning. The update equation is given by :

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}^k)$$

(b) For gradient-based learning it is suitable to have the gradient of the loss function as a large value and predictable value (ie constant gradient: linear behavior) so that we can have larger learning rate.

Such a behavior of the gradient can be achieved by pairing up softmax layer as the output layer and using a cross-entropy loss. This pairing helps in learning because the the log function in cross-entropy loss undoes the exponential in the softmax layer and helps avoiding saturation of the gradient. However, if we pair softmax layer with squared error loss then the gradient quickly saturates and we wont learn a good solution.

4 Problem 4: Cross-Entropy

(a) Given: Data $\mathcal{D} = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), ..., (\boldsymbol{x}_n, \boldsymbol{y}_n)\} \subset \mathcal{X} \times \mathcal{Y}$ sampled from p_{data}. We want to estimate p_{data} using a model with parameters $\boldsymbol{\theta}$.

The Maximum likelihood estimation is given by:

$$
\begin{aligned}
\boldsymbol{\theta}^* &= \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, p_{model}(\boldsymbol{y}_1, ..., \boldsymbol{y}_n | \boldsymbol{x}_1, ..., \boldsymbol{x}_n, \boldsymbol{\theta}) \\
&\text{Assuming independent samples} \\
&= \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \prod_{i=1}^{n} p_{model}(\boldsymbol{y}_i | \boldsymbol{x}_i, \boldsymbol{\theta}) \\
&= \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \log(\prod_{i=1}^{n} p_{model}(\boldsymbol{y}_i | \boldsymbol{x}_i, \boldsymbol{\theta})) \\
&= \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i=1}^{n} \log(p_{model}(\boldsymbol{y}_i | \boldsymbol{x}_i, \boldsymbol{\theta})) \\
&= \underset{\boldsymbol{\theta}}{\operatorname{argmin}} - \sum_{i=1}^{n} \log(p_{model}(\boldsymbol{y}_i | \boldsymbol{x}_i, \boldsymbol{\theta})) \\
&= \underset{\boldsymbol{\theta}}{\operatorname{argmin}} - \frac{1}{n} \sum_{i=1}^{n} \log(p_{model}(\boldsymbol{y}_i | \boldsymbol{x}_i, \boldsymbol{\theta})) \\
&\text{As } n \to \infty \text{ the above sum becomes the Expectation function} \\
&\simeq \underset{\boldsymbol{\theta}}{\operatorname{argmin}} - \mathbb{E}_{\boldsymbol{x} \sim p_{data}} [\log(p_{model}(\boldsymbol{y}_i | \boldsymbol{x}_i, \boldsymbol{\theta}))] \\
&= H(p, q)
\end{aligned}
$$

(b) Output units which whose output can be interpreted as a probability will work well with cross-entropy loss. For example Sigmoid and Softmax output units will work well with cross-entropy loss as they give a probability value.

5 Problem 5: Sigmoid and Softmax

(a) The Sigmoid ($\sigma(z)$) and Softmax functions are given by:

$$
\sigma(z) = \frac{1}{1 + \exp(-z)} \quad \text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}
$$

The output of the sigmoid function ($\sigma(z)$) can be interpreted as a conditional probability

distribution ($p(y|x)$) for a bernoulli random variable, which means it can be used as probability function for a binary classification problem.

The output of the Softmax function can be interpreted as a conditional probability distribution ($p(y|x)$) for a multinoulli random variable using one-hot encoding for each class. This means the Softmax function can be used as a probability function for multi-class classification problem.

Since both these functions give a conditional probability function they can be directly used in the cross-entropy loss function.

(b) The Sigmoid and Softmax output units can saturate easily. Therefore when paired with a squared error loss, the loss function will also quickly saturate. However when paired with a cross-entropy loss the log() function in undoes the exponential in of the output unit and we get a consistent gradient which wont saturate. This is good for gradient based learning as we will have a consistent gradient always which means the learning will be always possible till we get the best solution. For exmaple the cross-entropy loss for softmax is given by :

$$\begin{aligned}
\mathcal{L}(z) &= -\sum_i y_i \log(\text{softmax}(z_i)) \\
&= -\sum_i y_i \log(\frac{\exp(z_i)}{\sum_j \exp(z_j)}) \\
&= -\sum_i (z_i - \sum_j \exp(z_j)) \\
&\simeq -\sum_i z_i - \max(z) \qquad (2)
\end{aligned}$$

From Eq 2 we can see that the first term in the loss is a inear term dependent on z_i, even if the second term ($\max(z)$) saturates, the gradient of the loss will have a contribution from the first term always which means that we can still learn.

6 Problem 6: Linear Activation Function

The main idea behind using the linear activation as a hidden layer is that not all input units for a hidden layer are class-discriminatory thus their size can be reduced by summarizing the input units using linear functions. Linear activation functions are useful as hidden layers for deep forward neural networks as they help in summarizing parameters which can reduce noise. They also help the training process as the number of weights get reduced compared to a fully connected layer.

7 Problem 7: ReLU Activation

(a) The ReLU activation function for an input $\mathbf{z}$ is given by:

$$\text{ReLU}(\boldsymbol{z}) = \max\{0, z_i\}\boldsymbol{e}_i$$

Where $\boldsymbol{e}_i$ is the i^{th} standard basis vector. To state simply the ReLU() function boils down to a element-wise max function. The output of ReLU() for a single element is given by:

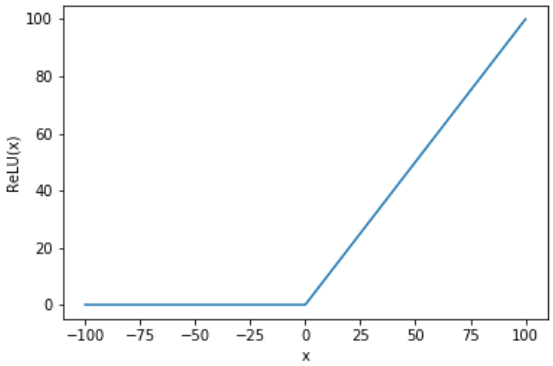


Figure 3: ReLU(z_i)

The gradient for ReLU() function is zero when z is negative and is a consistent value when z is positive which makes it a good choice for gradient-based learning.

(b) ReLU() can be generalized as:

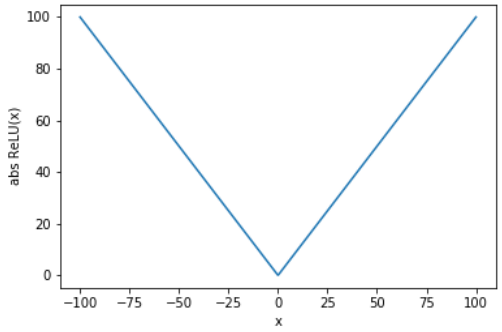
$$f(\boldsymbol{z}, \alpha) = (\max\{0, z_i\} + \alpha \min\{0, z_i\})\boldsymbol{e}_i$$

These generalizations are designed to overcome the drawback of regular ReLU() which cannot learn using gradient-based methods on examples for which the activation is zero.
Some examples of generalizations of ReLU() are:

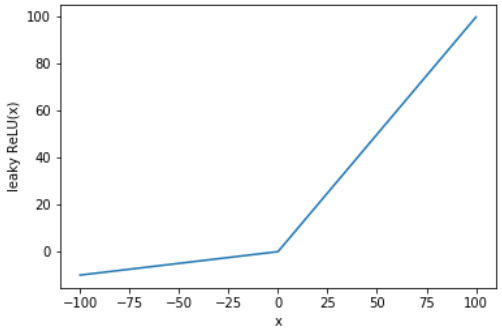
- Absolute value ReLU(): $\alpha = -1$, behavior show in figure 4(a)
  This activation function is used in applications of object recognition in images where we want to find features which are invariant to reversal of polarity of illumination as we will always have a consistent gradient when using absolute value ReLU().

- Leaky ReLU(): α =small positive value, behavior show in figure 4(b)
  This activation function is used to allow for a chance to activate a unit later in training or to make a more firm belief that it should not be active.

- Parametric ReLU(): α is a learn-able parameter
  Allows more flexibility for the neural network to choose the value of α so as to learn a better classifier, but this also adds to computational cost of the network.



(a) Absolute value ReLU



(b) Leaky ReLU for $\alpha = 0.1$

Figure 4: ReLU() generalizations

(c) The Maxout function generalizes ReLU() further. Instead of applying elemen-wise $max(0, z_k)$, maxout function divide z into groups of k values. Each maxout unit then outputs the maximum element of one of these groups:

$$maxout(\boldsymbol{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j$$

Where $\mathbb{G}^{(i)}$ is the set of indices into the inputs for group $i, \{(i-1)k+1, ...ik\}$. This helps by providing a way of learning piece-wise linear activation function upto k pieces but the disadvantage is that each maxout unit now requires k weights instead of just one. Thus it is computationally more expensive.

8 Problem 8: Universal Approximation

(a) The Universal Approximation theorem states that a feed forward network with a linear output layer and at least one hidden layer with an activation function that saturates for large positive and negative values (ie "squashing" activation function), can approximate any Borel Measurable function from a finite dimensional space to another with desired non-zero error upper bound, provided that the network is given enough hidden units.

(b) The universal approximation theorem generalizes to deep neural nets by stating that there always exists a large multi-layer perceptron (MLP) which will be able to learn the exact function we want for the classifier, however it does not state how large/deep the network should be in order to learn such a function. Also, even if the MLP is able to represent the function, learning can fail still fail due to optimization method or due to overfitting to training data.

(c) Choosing a deeper network over a shallow network requires smaller number of parameters (for a fixed amount of "memory budget" ie number of units).

A deeper network with lots of layers imposes the belief that the function we want to learn is com-

posed of several simpler functions, this is to say that for an object detection problem a deeper network will first try to identify points, then edges connecting to points, then shapes from edges and so on to identify the object which helps in generalization of a deeper network. A shallow network of say a single layer will try to find a single feature to describe the whole object and would thus not generalize well.

Since a deeper network has the implied belief of composition of functions, these beliefs are equivalent to constraints for the optimization problem and thus a deeper network is difficult to train.

9 Problem 9: Back-propagation

(a) Given:

$$f : \mathbb{R}^n \to \mathbb{R}^m \quad g : \mathbb{R}^m \to \mathbb{R}^1$$
$$h = g \circ f$$
$$\text{ie } h(\boldsymbol{x}) = g(\boldsymbol{z})$$
$$\text{where } \boldsymbol{z} = f(\boldsymbol{x})$$

To Find: $\nabla_{\boldsymbol{x}} h(\boldsymbol{x})$

Solution:

$$\begin{aligned}
\nabla_{\boldsymbol{x}} h(\boldsymbol{x}) &= \nabla_{\boldsymbol{x}} g(\boldsymbol{z}) \\
&= \frac{\partial g}{\partial x_i} = \frac{\partial g}{\partial z_j}\frac{\partial z_j}{\partial x_i} = \frac{\partial g}{\partial z_j}\frac{\partial f_j}{\partial x_i} \\
&= (\nabla_{\boldsymbol{z}} g)^T (\nabla_{\boldsymbol{x}} \boldsymbol{f}) \\
&= J^T \nabla_{\boldsymbol{z}} g \\
&\text{where } J = \nabla_{\boldsymbol{x}} f \text{ is the Jacobian of } f(\boldsymbol{x})
\end{aligned}$$

$$\Rightarrow \nabla_{\boldsymbol{x}} h(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & .... & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & .... & \frac{\partial f_2}{\partial x_n} \\ . & . & .... & . \\ . & . & .... & . \\ . & . & .... & . \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & .... & \frac{\partial f_m}{\partial x_n} \end{bmatrix}^T \begin{bmatrix} \frac{\partial g}{\partial z_1} \\ \frac{\partial g}{\partial z_2} \\ . \\ . \\ . \\ \frac{\partial g}{\partial z_m} \end{bmatrix}$$

(b) Multi-Layer network defined using composition:

Given: $J : \mathbb{R}^d \to \mathbb{R}$

$$J(u_1, u_2, ...u_d) = f_l \circ f_{l-1} \circ .... \circ f_1(w_1, u_2, ..., u_d) \tag{3}$$

Where $f_j : \mathbb{R}^{n_j} \to \mathbb{R}^{n_{j+1}}$ and $n_1 = d$, $n_{l+1} = 1$

(i) Back Propagation to compute $\nabla J(u_1, u_2, ...u_d)$

First we need to evaluate the variables in the graph $\mathcal{G}$ using forward propagation. The algorithm for forward propagation is given by:

Algorithm 1 Forward Propagation: Computes the output $J = u^{(n)}$ using the computational graph $\mathcal{G}$ defined by Eq 3, Where each node in $\mathcal{G}$ computes a numerical value $u^{(i)}$ by applying an activation function $f^{(i)}$ to the set of arguments $\mathbb{A}^{(i)}$ that comprises the values of previous nodes $u^{(j)}, j < i,$ with $j \in Pa(u^{(i)})$

for $i = 1, ..., d$ **do**
  $u^{(i)} \leftarrow x_i$; `// Initialization`
end for
for $i = d + 1, ..., n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} | j \in Pa(u^{(i)})\}$; `// Collection of all nodes which are parents of` i^{th} `node`
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$; `// apply the activation function`
end for
return $u^{(n)}$

After evaluation of all the variables using forward propagation as per Algorithm 1, we can now carry out back propagation where we would use the values computed during forward propagation.

For Backward propagation we construct the graph $\mathcal{B}$ which has the same structure as $\mathcal{G}$ except that the direction of the edges becomes opposite and we do computation from top to bottom. The nodes in $\mathcal{B}$ correspond to the partial derivative of the final variable with respect to the variable that node belongs to, ie $\frac{\partial u^{(n)}}{\partial u^{(i)}}$. We then evaluate the gradient using algorithm for backward propagation as follows:

Algorithm 2 Backward Propagation: Initialize `grad_table`, a data structure to store the final derivatives. The entry `grad_table`$[u^{(i)}]$ stores the computed values of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.

`grad_table`$[u^{(n)}] \leftarrow 1$; `// Initialization`
for j = n-1 down to 1 **do**
  `grad_table`$[u^{(j)}] \leftarrow \sum_{i:j \in Pa(u^{(i)})}$ `grad_table`$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$
end for
return $\{$ `grad_table`$[u^{(i)}] \quad | \quad i = 1, ..., d \}$

(ii) The computational cost of the Algorithm 2 is proportional to the number of edges in the

graph $\mathcal{G}$. As the computation graph is a directed acyclic graph it has at most $O(n^2)$ edges. And for each edge we perform a Jacobian-vector product.

(b) Multi-Layer Feed-Forward Network $f : \mathbb{R}^d \to \mathbb{R}^0$ defined recursively as:

$$f(\boldsymbol{x}) = \boldsymbol{h}^{(l)}$$
$$\boldsymbol{h}^{(k)} = \sigma(\boldsymbol{a}^{(k)}), \boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-1)}, k \in \{1, 2, ..., l\}$$

The loss function does not have the regularization term ie $\Omega(\boldsymbol{w}) = 0$.

(i) Computation of gradient of loss function $\nabla J = \nabla \mathcal{L}$

Prior to computation of gradient of the loss function we need to evaluate the activations $a^{(k)}$ for each layer k. These values will then be used for back propagation. We carry out the forward propagation using the following algorithm:

Algorithm 3 Forward Propagation: In the forward propagation we compute the loss $\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$. We initialize the weights $\boldsymbol{W}^{(i)}$ and biases $\boldsymbol{b}^{(i)}$ for all $i \in \{1, 2, ..., l\}$. For any input $\boldsymbol{x}$ with target output $\boldsymbol{y}$. The Forward propagation computation is given by:

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for k=1,2,....,l **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-1)}$
  $\boldsymbol{h}^{(k)} = \sigma(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = \mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$

After forward computation we store the computed values of activations $\boldsymbol{a}^{(k)}$ for each layer k and then can carry out back propagation as follows:

Algorithm 4 Back Propagation: We compute the gradients with respect to $\boldsymbol{a}^{(k)}$ starting from the output layer and going backwards to the first hidden layer. From these gradients we get to compute the gradient of the loss function with respect to weights and biases. For computation we allocate a variable $\boldsymbol{g}$ which stores the intermediate gradient value.

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} \mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for k=l,l-1,....1 **do**
  We convert the gradient on the layer's output into a gradient on the pre-non-linearity activation.
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot \sigma'(\boldsymbol{a}^{(k)})$
  We then compute the gradients on weights and biases for the layer as:
  $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g}$
  $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\boldsymbol{h}^{(k-1)T}$
  We then propagate the gradients for the next lower-level hidden layer's activations:
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} = \boldsymbol{W}^{(k)T}\boldsymbol{g}$
end for

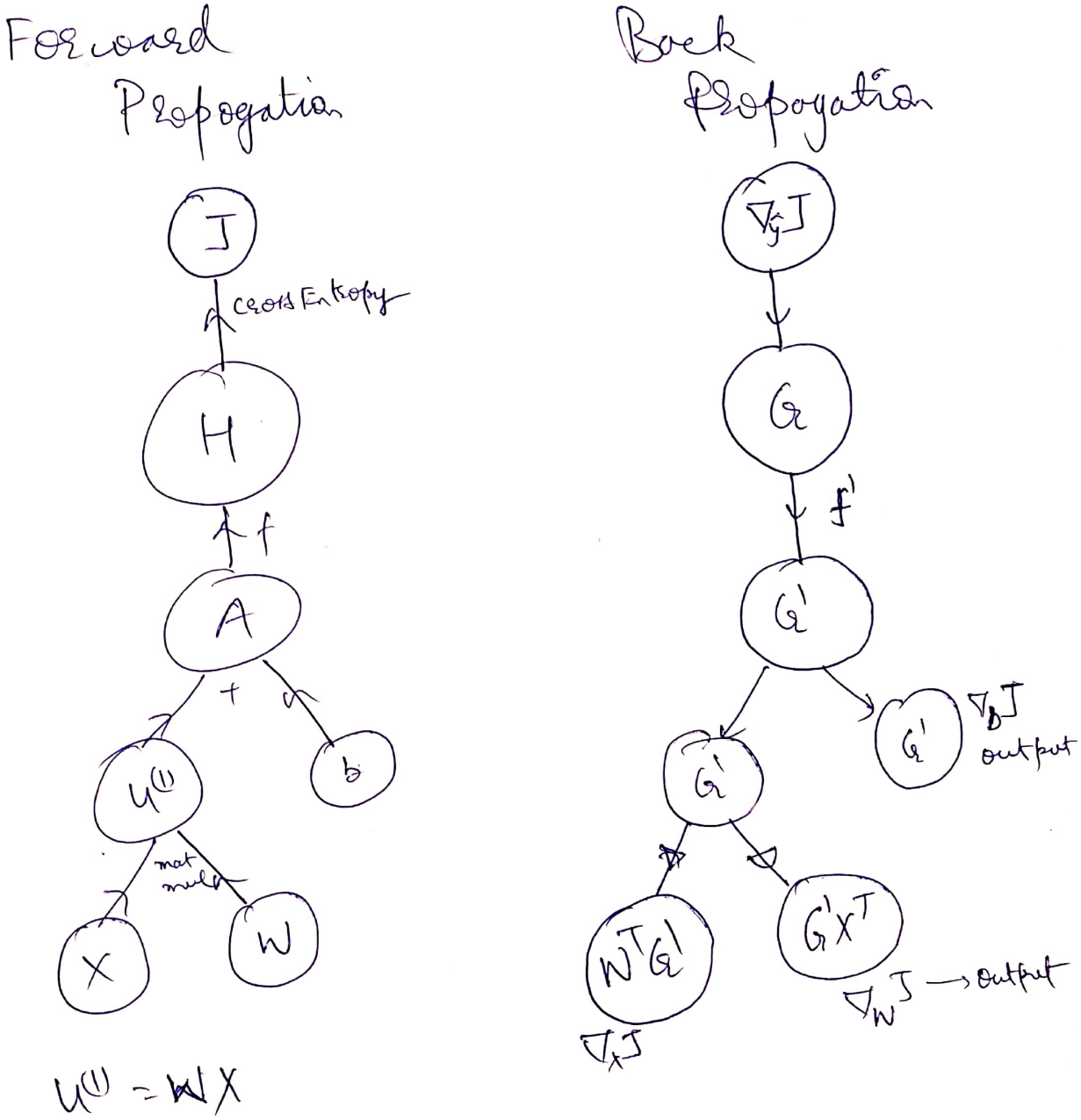
In the above computation, we need to use σ' which will depend on the type of hidden layer operation. For example, for ReLU() activation σ' is given by:

$$\sigma'(\boldsymbol{a}^{(k)}) = \frac{\partial \sigma'(a_i^{(k)})}{\partial a_i} \boldsymbol{e_i}$$

$$\text{where } \frac{\partial \sigma'(a_i^{(k)})}{\partial a_i} = \begin{cases} 1, & \text{if} \quad a_i^{(k)} > 0 \\ 0, & \text{otherwise} \end{cases}$$

Thus after carrying out back-propagation as per Algorithm 4 we can compose the gradients $\nabla_{\boldsymbol{b}^{(k)}} J \quad \text{and} \quad \nabla_{\boldsymbol{W}^{(k)}} J$ for each layer k to make the final gradient ∇J.

(ii) The Computational graph for forward and back propagation for a single layer network will be as follows:



(d) Let's take an example to explain the trade-off between computation and storage cost for computing the gradient. Let the feed-forward network be defined as:

$$
\begin{aligned}
& f : \mathbb{R} \to \mathbb{R} \\
& w \in \mathbb{R} \\
& x = f(w), y = f(x), z = f(y) \\
\text{Then gradient is: } & \frac{\partial z}{\partial w} \\
& = f'(y)f'(x)f'(w) && (4) \\
& = f'(f(f(w)))f'(f(w))f'(w) && (5)
\end{aligned}
$$

The gradient $\frac{\partial z}{\partial w}$ can be computed using two methods as per Eq 4 or Eq 5. While the method Eq 4 requires fewer computation of functions f it requires us to store the computed values of the variables x, y computed during the forward propagation. Whereas for method using Eq 5, we need lots of function evaluations but do need to store the values of variables x, y. From this simple example we can see that based on our choice/resources available we will have to choose between a computationally efficient or memory efficient algorithm for computing the gradient.

The basic backpropagation algorithm is a computationally efficient algorithm but requires memory to store the values of intermediate variables.

10 Problem 10: Back-propagation Implementation

(a) The biggest difference between the method of gradient computation in Tensorflow and Torch is that while in Torch the gradient is computed using symbol-to-number differentiation approach, in Tensorflow the gradient is computed using symbol-to-symbol based approach.

(b) In Tensorflow the `op.bprop()` method is used in the back-propagation algorithm to evaluate the derivative of the activation function using the operation's `bprop` rules with right arguments. The `op.bprop(inputs,X,G)` method should return the following:

$$\texttt{op.bprop(inputs,X,G)} = \sum_i (\nabla_{\boldsymbol{X}} \text{ op.f(inputs)}_i) G_i$$

Where `input` is a list of inputs that are supplied to the operation `op.f()` which implements the activation function f, $\mathbf{X}$ is the input whose gradient we wish to compute and $\mathbf{G}$ is the gradient on the output of operation. For example for matrix multiplication operation:

$$\texttt{op.f}(\boldsymbol{X}) = \boldsymbol{W}\boldsymbol{X}$$

The `op.bprop(inputs,X,G)` is given by:

$$\begin{aligned}
\texttt{op.bprop(inputs,X,G)} &= \sum_i (\nabla_{\boldsymbol{X}} \text{ op.f(inputs)}_i) G_i \\
&= \boldsymbol{G} \nabla_{\boldsymbol{X}} (\boldsymbol{W}\boldsymbol{X}) \\
&= G_{ij} \frac{\partial W_{ik} X_{kj}}{\partial X_{mn}} = G_{ij} W_{ik} \delta_{km} \delta_{jn} \\
&= G_{in} W_{im} \boldsymbol{e}_m \otimes \boldsymbol{e}_n \\
&= \boldsymbol{W}^T \boldsymbol{G}
\end{aligned}$$