Neural Networks Homework 6

Sayeh Jarollahi (7073520, saja00006@stud.uni-saarland.de)
Mahsa Amani (7064006, maam00002@stud.uni-saarland.de)

December 4, 2024

Exercise 6.1

Proof. 1)
a) $\sigma(-x) = 1 - \sigma(x)$
Using sigmoid function definition and substituting x with $-x$ we have:

$$\sigma(x) = \frac{1}{1+e^{-x}} \implies \sigma(-x) = \frac{1}{1+e^{x}}$$

Now, we calculate the right hand side of the equation, $1 - \sigma(x)$:

$$1 - \sigma(x) = 1 - \frac{1}{1+e^{-x}} = \frac{(1+e^{-x}) - 1}{1+e^{-x}} = \frac{e^{-x}}{1+e^{-x}}$$

We can simplify $\sigma(-x)$ by multiplying both nominator and denominator into e^{-x}:

$$\sigma(-x) = \frac{1}{1+e^{x}} = \frac{e^{-x}}{e^{-x}+1}$$

So we can conclude:

$$\sigma(-x) = 1 - \sigma(x)$$

b) $\tanh(x) = 2\sigma(2x) - 1$
Using hyperbolic tangent function's definition and substituting x with $2x$ in the sigmoid function, we have:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}, \sigma(2x) = \frac{1}{1+e^{-2x}}$$

By multiplying both nominator and denominator in $\sigma(2x)$ to e^{x}:

$$\sigma(2x) = \frac{e^{x}}{e^{x} + e^{-x}}$$

Now for the right hand side of the equation we have:

$$2\sigma(2x) - 1 = 2 \cdot \frac{e^{x}}{e^{x} + e^{-x}} - 1 = \frac{2e^{x} - (e^{x} + e^{-x})}{e^{x} + e^{-x}} = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Which is equal to $\tanh(x)$.
c) $\frac{d}{dx}\sigma(x) = (1-\sigma(x))\cdot\sigma(x) = \sigma(-x)\cdot\sigma(x)$
Using sigmoid function's definition, we take its derivative and simplify it with chain rule:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\frac{d}{dx}\sigma(x) = \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) = -\frac{-e^{-x}}{(1+e^{-x})^2} = \frac{e^{-x}}{(1+e^{-x})^2}$$

Now calculate the middle term in the equation, $(1-\sigma(x))\cdot\sigma(x)$:

$$\sigma(x) = \frac{1}{1+e^{-x}}, \quad 1-\sigma(x) = \frac{e^{-x}}{1+e^{-x}}$$

$$(1-\sigma(x))\cdot\sigma(x) = \frac{e^{-x}}{1+e^{-x}}\cdot\frac{1}{1+e^{-x}} = \frac{e^{-x}}{(1+e^{-x})^2}$$

Which is equal to left hand side of the equation:

$$\frac{d}{dx}\sigma(x) = (1-\sigma(x))\cdot\sigma(x)$$

Finally, using part (a), $\sigma(-x) = 1-\sigma(x)$, we can conclude the right hand side of the equation also holds true.
d) $\frac{d}{dx}\tanh(x) = 4(1-\sigma(2x))\cdot\sigma(2x)$
From equation (b) we know $\tanh(x) = 2\sigma(2x)-1$. So we take its derivative with respect to x:

$$\frac{d}{dx}\tanh(x) = \frac{d}{dx}\left(2\sigma(2x)-1\right) = 2\cdot\frac{d}{dx}\sigma(2x) = 2\cdot\frac{d}{dx}\left(\frac{1}{1+e^{-2x}}\right) = 2\cdot\frac{2e^{-2x}}{(1+e^{-2x})^2} = \frac{4e^{-2x}}{(1+e^{-2x})^2}$$

For the right hand side equation we have:

$$4(1-\sigma(2x))\cdot\sigma(2x) = 4(1-\frac{1}{1+e^{-2x}})\frac{1}{1+e^{-2x}} = \frac{4e^{-2x}}{(1+e^{-2x})^2}$$

So we prove 2 sides are equal.
2)
Based on the question we have:

$$y_k(\mathbf{x},\mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

where $h(a) = \sigma(a)$.
Using equation (b) from part 1, we know $\tanh(x) = 2\sigma(2x)-1$. So:

$$h(a) = \sigma(a) = \frac{1+\tanh(a/2)}{2}$$

Now, we substitute this into the network definition:

$$h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) = \frac{1 + \tanh\left(\frac{1}{2}\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right)\right)}{2}$$

The output function becomes:

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} \frac{1 + \tanh(\frac{\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}}{2})}{2} + w_{k0}^{(2)}\right)$$

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\frac{1}{2}\sum_{j=1}^{M} w_{kj}^{(2)} + \frac{1}{2}\sum_{j=1}^{M} w_{kj}^{(2)} \tanh(\frac{1}{2}(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)})) + w_{k0}^{(2)}\right)$$

Now, to use $\tanh(a)$ as the hidden activation, we redefine the network parameters as follows:

$$w_{ji}^{(1)\prime} = \frac{1}{2} w_{ji}^{(1)}, \quad w_{j0}^{(1)\prime} = \frac{1}{2} w_{j0}^{(1)}, \quad w_{kj}^{(2)\prime} = \frac{1}{2} w_{kj}^{(2)}, \quad w_{k0}^{(2)\prime} = w_{k0}^{(2)} - \sum_{j=1}^{M} w_{kj}^{(2)}$$

Finally, for the new network with $\tanh(a)$ hidden activations we have:

$$y_k(\mathbf{x}, \mathbf{w}') = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)\prime} \tanh\left(\sum_{i=1}^{D} w_{ji}^{(1)\prime} x_i + w_{j0}^{(1)\prime}\right) + w_{k0}^{(2)\prime}\right)$$

3)
For the derivative of sigmoid function $\sigma(x)$ we have:

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x))$$

where $\sigma(x) \in (0, 1)$ for all $x \in \mathbb{R}$. If we substitute $\sigma(x)$ with t:

$$\frac{d}{dx}\sigma(x) = t(1 - t) = -t^2 + t = f(t)$$

By taking the derivative of $f(t)$:

$$f'(t) = -2t + 1$$

And finding its critical point when $f'(t) = 0$:

$$-2t + 1 = 0 \implies t = \frac{1}{2}$$

$$f\left(\frac{1}{2}\right) = \frac{1}{2}(1 - \frac{1}{2}) = \frac{1}{4}$$

As this function $f(t) = t(1-t)$ is symmetric around $t = \frac{1}{2}$ and reaches its maximum value of $\frac{1}{4}$ at $t = \frac{1}{2}$ and $t \in (0,1)$, $f(t) > 0$ for all t. So:

$$0 < \frac{d}{dx}\sigma(x) \leq \frac{1}{4}$$

For the derivative of hyperbolic tangent function $\tanh(x)$ we have:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$

We know $\tanh(x) \in (-1,1)$ for all $x \in \mathbb{R}$, so $\tanh^2(x) \in (0,1)$, and $1 - \tanh^2(x) \in (0,1)$, and finally:

$$0 < \frac{d}{dx}\tanh(x) \leq 1$$

Both sigmoid and tanh activation functions suffer from the **vanishing gradient problem**, where their derivatives become very small for large positive or negative inputs. This results in very small gradients during backpropagation in deep networks, hindering effective weight updates in earlier layers. To prevent this we can use the ReLU (Rectified Linear Unit) activation function, which is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

and its derivative is:

$$\frac{d}{dx}\text{ReLU}(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases}$$

With ReLU we can address the vanishing gradient problem because for all positive inputs, the gradient is constant and equal to 1 and it does not saturate for large positive inputs. Also we can use other variants of ReLU like Leaky ReLU, which allows a small gradient for $x < 0$, and defined as:

$$\text{Leaky ReLU}(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{if } x \leq 0 \end{cases}$$

where $\alpha > 0$ and has a low value like 0.01.

4)
The softmax function for a vector $\mathbf{y} = (y_1, y_2, \ldots, y_n)$ is defined as:

$$\text{softmax}(y)_i = \frac{e^{y_i}}{\sum_{j=1}^{n} e^{y_j}}, \quad i = 1, 2, \ldots, n$$

Now by adding c to $\mathbf{y}$, where $c = (c, c, \ldots, c)^T$ is a constant vector, we will have:

$$\text{softmax}(y + c)_i = \frac{e^{y_i + c}}{\sum_{j=1}^{n} e^{y_j + c}} = \frac{e^{y_i} \cdot e^{c}}{\sum_{j=1}^{n} e^{y_j} \cdot e^{c}} = \frac{e^{c} \cdot e^{y_i}}{e^{c} \cdot \sum_{j=1}^{n} e^{y_j}} = \frac{e^{y_i}}{\sum_{j=1}^{n} e^{y_j}} = \text{softmax}(y)_i$$

This proves that the softmax function is invariant with respect to the addition of a constant vector c.
The main numerical issue with the softmax function arises from the computation of the exponentials e^{y_i}. One is **Overflow** where for very large values of y_i, e^{y_i} can exceed the maximum representable value in floating-point arithmetic, causing numerical overflow. Another one is **Underflow** where for very small values of y_i, e^{y_i} can become so close to zero that it causes numerical underflow, leading to inaccurate results. To prevent these, we can use the invariance property of softmax and instead of directly computing softmax(y), subtract the maximum value in the vector y from all the values and then compute the softmax.

$$\text{softmax}(y)_i = \frac{e^{y_i - \max(y)}}{\sum_{j=1}^{n} e^{y_j - \max(y)}}$$

Setting the largest value to 0 ensures all other values are less than or equal to 0, preventing the exponentials $e^{y_i - \max(y)}$ from becoming too large and avoiding overflow. This also keeps smaller values of $y_i - \max(y)$ closer to 0, reducing the risk of underflow.
5)
Without activation functions, each layer performs only linear transformations, meaning that stacking multiple layers results in a single linear transformation. For example, if we have $y = W_2(W_1x+b_1)+b_2$, it simplifies to $y = (W_2W_1)x+(W_2b_1+b_2)$, which is equivalent to one linear layer. Therefore, a deep neural network without activation functions possesses the same expressiveness as a single-layer network, no matter how many layers are added. Activation functions introduce non-linearity, enabling neural networks to learn complex patterns that linear combinations cannot represent. This non-linearity allows the network to manipulate the input space, facilitating the separation and classification of intricate data patterns. □

Exercise 6.2

Proof. a) Output Function: Sigmoid
Loss Function: Binary Cross-Entropy
b) Output Function: Softmax
Loss Function: Categorical Cross-Entropy
c) Output Function: Linear
Loss Function: Mean Squared Error (MSE)
d) Output Function: Sigmoid, In multi-label classification, each label is independent, requiring each output neuron to predict the probability of the label's presence. The sigmoid function is ideal for this, as it independently outputs probabilities for each label within the range [0,1].
Loss Function: Binary Cross-Entropy, it is suitable for multi-label classification as it treats each label as an independent binary classification problem, computing the loss for each label separately.
e) Output Function: Softmax, For next-word prediction, the model outputs a probability distribution over the vocabulary, with Softmax ensuring the probabilities sum to 1 and enabling the prediction of the most likely word.

Loss Function: Categorical Cross-Entropy, Cross-entropy loss is ideal for multi-class predictions, such as next-word prediction, as it compares the predicted probability distribution from softmax with the true label, which is the actual next word. □

Exercise 6.3

Proof. a)
Our inputs are $x_1 = 3, x_2 = 1, x_3 = -1$ and the weight matrix for the first hidden layer ($W^{(1)}$) is:

$$W^{(1)} = \begin{bmatrix} 2 & -1 \\ 1 & -3 \\ -2 & 5 \end{bmatrix}$$

So for h_1 and h_2 we have:

$$h_1 = \text{ReLU}\left(2(3) + 1(1) + (-2)(-1)\right) = \text{ReLU}(6 + 1 + 2) = \text{ReLU}(9) = 9$$

$$h_2 = \text{ReLU}\left((-1)(3) + (-3)(1) + 5(-1)\right) = \text{ReLU}(-3 - 3 - 5) = \text{ReLU}(-11) = 0$$

Thus:

$$h_1 = 9,\ h_2 = 0$$

Now, we compute o_1 an o_2, using $o_k = \sum_{j=1}^{2} W_{jk}^{(2)} h_j$, where the weight matrix for the output layer ($W^{(2)}$) is:

$$W^{(2)} = \begin{bmatrix} 6 & 2 \\ -1 & 8 \end{bmatrix}$$

$$o_1 = 6(9) + (-1)(0) = 54 + 0 = 54$$

$$o_2 = 2(9) + 8(0) = 18 + 0 = 18$$

Thus:

$$o_1 = 54,\ o_2 = 18$$

Now, we apply the softmax activation function over them:

$$p_1 = \frac{e^{54}}{e^{54} + e^{18}} = \frac{1}{1 + e^{-36}}$$

$$p_2 = \frac{e^{18}}{e^{54} + e^{18}} = \frac{e^{-36}}{1 + e^{-36}}$$

Since e^{-36} is very small (near 0), we approximate $p_1 \approx 1$ and $p_2 \approx 0$.
b)
If this is a binary classification problem, the predicted class corresponds to the class with the highest softmax probability which is p_1. So its respective class would be the predicted class. □

Exercise 6.4

Proof. a) Batch Normalization is used for optimization and normalization of the layers in the neural networks. It prevents the change in distribution of inputs of each layer in a neural network when the parameters of the previous layers update during training. It also can stablize the input distribution of layers to let the network use higher learning rates.

b) No, it addds flexibility and learability to the network by learninig the optimal scale and shift for the normalized activations. Normalization ensures stable and smooth optimization by keeping activations in a consistent range, helping gradients flow effectively during training. Re-scaling ensures that the model isn't overly constrained by normalization and can still learn complex representations.

c) With a batch size of 1, the statistics are computed from a single sample, meaning that the batch mean equals the sample's value and the variance becomes zero, resulting in a division by zero or instability. During inference, the BatchNorm layer no longer uses the statistics of the incoming batch. Instead, it relies on running statistics (mean and variance) that were computed and updated during training.

d) BatchNorm is successful because it smooths the optimization landscape, making gradients more stable and predictive, which allows for faster, more reliable training with larger learning rates and reduced sensitivity to hyperparameter choices. It also, BatchNorm is successful because it stabilizes the loss function and its gradients, improving their predictiveness and smoothness, which enhances training efficiency and robustness across various directions in the optimization landscape.

□

Exercise 6.5

Proof. a) Weight space symmetry can hinder the optimization process because in a neural network, permuting the weights of two neurons in the same layer does not change the output of the network so the network has many local minimas. Also flat regions created by symmetric weight can cause small gradients and slow training.

b) They can always converge under one condition: if the learning rate is adjusted correctly. If not, it can lead to divergence and never get to the optimum point.

c) We consider the function:

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c$$

where we consider A = I. By newton method in one step we get the result b for x. However in the gradient descent we need more steps to reach this value.

d) Newton method needs the Hessian matrix, however in the big neural networks with millions of parameters, this is not feasible to calculate the second derivative. Also, the

inversion of hesian is needed which makes it more complicated and not possible for large models. □

Exercise 6.7

Proof. a) The critical points that pose a major challenge in high-dimensional non-convex optimization, as highlighted in the paper, are saddle points rather than local minima.Saddle points are more abundant in high-dimensional data. Saddle points are often surrounded by high-error plateaus. These regions can dramatically slow down optimization processes, as algorithms like gradient descent or quasi-Newton methods struggle to escape them efficiently.

b) Gradient descent behaves differently near saddle points and local minima due to the curvature of the error surface. At saddle points, gradient descent is repelled in directions of negative curvature; however, this repulsion can be slow when the negative eigenvalues of the Hessian are small, resulting in prolonged stagnation in the vicinity of the saddle point. While gradient descent's instability at saddle points can eventually push it away, the process is often inefficient due to the flatness of the surrounding landscape, leading to slow convergence and difficulty in escaping these regions. (Based on Introduction, page 2)

c) The limitation of Newton's method that makes it unsuitable for escaping saddle points lies in its treatment of directions with negative curvature. Newton's method rescales gradient steps by the inverse of the Hessian's eigenvalues to accelerate convergence. However, when encountering a negative eigenvalue, this rescaling inverts the direction of the gradient along the corresponding eigenvector. Instead of moving away from a saddle point, the method steps toward it, effectively turning the saddle point into an attractor. This fundamental issue causes Newton's method to converge to unstable critical points like saddle points, where it cannot escape, unlike gradient descent, which may eventually be repelled in directions of negative curvature. (based on page 5) □