Different Types of Activation Units – Assignment

1. Linear Activations

Consider a single-layer neural network represented as $g(\mathbf{x}) = \mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$, where $\mathbf{x} \in \mathbb{R}^d$ is the input, $\mathbf{W}_1 \in \mathbb{R}^{d' \times d}, \mathbf{W}_2 \in \mathbb{R}^{D \times d'}, \mathbf{b}_1, \mathbf{b}_2$ are parameters of this neural network, and $\sigma : \mathbb{R} \to \mathbb{R}$ is an activation function applied element-wise to each input entries.

(a) Expressiveness of ReLU nets. If σ is the ReLU activation function, can this neural network approximate $f(x) = x$ in $[0, 1]$? If it is possible, give your $d', \mathbf{W}_1, \mathbf{W}_2$, such that $\sup_{x\in[0,1]} |f(x) - x^2| \leq 0.04$; if it is impossible, give your proof.

The result is obviously yes because of the universal approximation theorem. There are multiple ways to construct an example. Here we give a relatively simple way by iteratively subtract $\max\{\alpha_i(x - \beta_i), 0\}$ from x^2.

Because $f(0) = 0^2 = 0$, we construct $\beta_1 = 0$. We want to choose α_1 and ensure that $f(x) - \max\{\alpha_1(x - \beta_1), 0\} \geq -0.04$, which is equivalent to $x^2 - \alpha_1 x \geq -0.04$ for all $x \geq 0$, so $\alpha_1 \leq 0.2$. We choose $\alpha_1 = 0.2$ in this case. So $f_1(x) = f(x) - \max\{0.2x, 0\} = x^2 - 0.2x$ $(x \in [0, 1])$

For $x \leq 0.2$, we are done because $-0.04 \leq f_1(x) \leq 0$ within this interval, so we set $\beta_2 = 0.2$ and consider $x \in [0.2, 1]$. We want $f_1(x) - \alpha_2(x - 0.2) \geq -0.04$ for $x \geq 0.2$. The condition is $(0.1 + 0.5\alpha_2)^2 \leq 0.04 + 0.2\alpha_2$, where the solution is $\alpha_2 \leq 0.6$, so we choose $\alpha_2 = 0.6$. So $f_2(x) = f_1(x) - \max\{0.6(x - 0.2), 0\}$

Now let us consider $x \geq 0.6$, where $f_2(x) = x^2 - 0.2x - 0.6(x - 0.2) = x^2 - 0.8x + 0.12$. Set $\beta_3 = 0.6$ and we want $x^2 - 0.8x + 0.12 - \alpha_3(x - 0.6) \geq -0.04$. The condition is $(0.4 + 0.5\alpha_3)^2 \leq 0.16 + 0.6\alpha_3$, so $\alpha_3 \leq 0.8$. Therefore, $f_3(x) = f_2(x) - \max\{0.8(x - 0.6), 0\}$

We find that $f_3(1) = 0 \leq 0.04$, so the iteration finished. To summarize, our approximation is $f(x) - f_3(x) = \max\{0.2x, 0\} + \max\{0.6(x - 0.2), 0\} + \max\{0.8(x - 0.6), 0\}$, which can be expressed by a ReLU net with parameters: $\mathbf{W}_1 = (0.2, 0.6, 0.8)^T, \mathbf{b}_1 = (0, -0.12, -0.48)^T, \mathbf{W}_2 = (1, 1, 1)$ and $d' = 3$. As is shown in Figure 1.

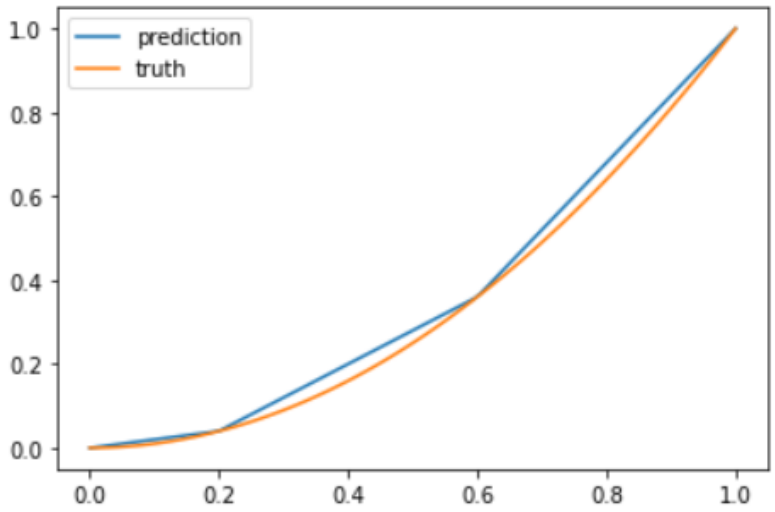


Figure 1. Approximating $x \mapsto x^2$ using a single-layer ReLU net.

(b) Expressiveness of NNs using linear activations. If we replace the activation functions in (a) by the identity function $\sigma(x) = x$, can this neural network approximate the same function in $[0, 1]$ uniformly by 0.04? Explain why.

The result is no. If the activation function is identity, then the model is $g(x) = \mathbf{W}_2\mathbf{W}_1\mathbf{x} + \mathbf{W}_2\mathbf{b}_1 + b_2$, which is a linear mapping of x, and it can be expressed as $g(x) = \alpha x + \beta$. Intuitively, we want to find an approximation of $x \mapsto x^2$ using a linear function. Let us prove that it is impossible to get a uniform approximation error below 0.04.

$\exists \alpha, \beta, \text{s.t. } \forall x \in [0,1], |f(x) - g(x)| = |x^2 - \alpha x - \beta| \leq 0.04$, if and only if $\exists \alpha, \text{s.t. } \max_{x\in[0,1]}\{x^2 - \alpha x\} - \min_{x\in[0,1]}\{x^2 - \alpha x\} \leq 0.08$. Now let us consider $h(x) = x^2 - \alpha x$
We need $|h(1) - h(0)| \leq 0.08$, so $|1 - \alpha| \leq 0.08 \implies \alpha \in [0.92, 1.08]$. So the minimal of h in $[0,1]$ is $h_{\min} = h(0.5\alpha) = -0.25\alpha^2$. We need $h(0) - h_{\min} \leq 0.08$, so $0.25\alpha^2 \leq 0.08$, which is impossible for $\alpha \in [0.92, 1.08]$. Therefore, we cannot find such a pair of α, β such that $\forall x \in [0,1], |f(x) - g(x)| \leq 0.04$.

2. Derivation of Different Activation Functions

Derive the derivative of the following activation functions.

(a). ReLU

$$f(z) = \begin{cases} z & if \quad z \geq 0, \\ 0 & else. \end{cases}$$

$$f'(z) = \begin{cases} 1 & if \quad z \geq 0, \\ 0 & else. \end{cases}$$

(b). Sigmoid

$$f(z) = \frac{1}{1 + e^{-z}}$$

$$f'(z) = f(z)(1 - f(z))$$

(c). tanh

$$f(z) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

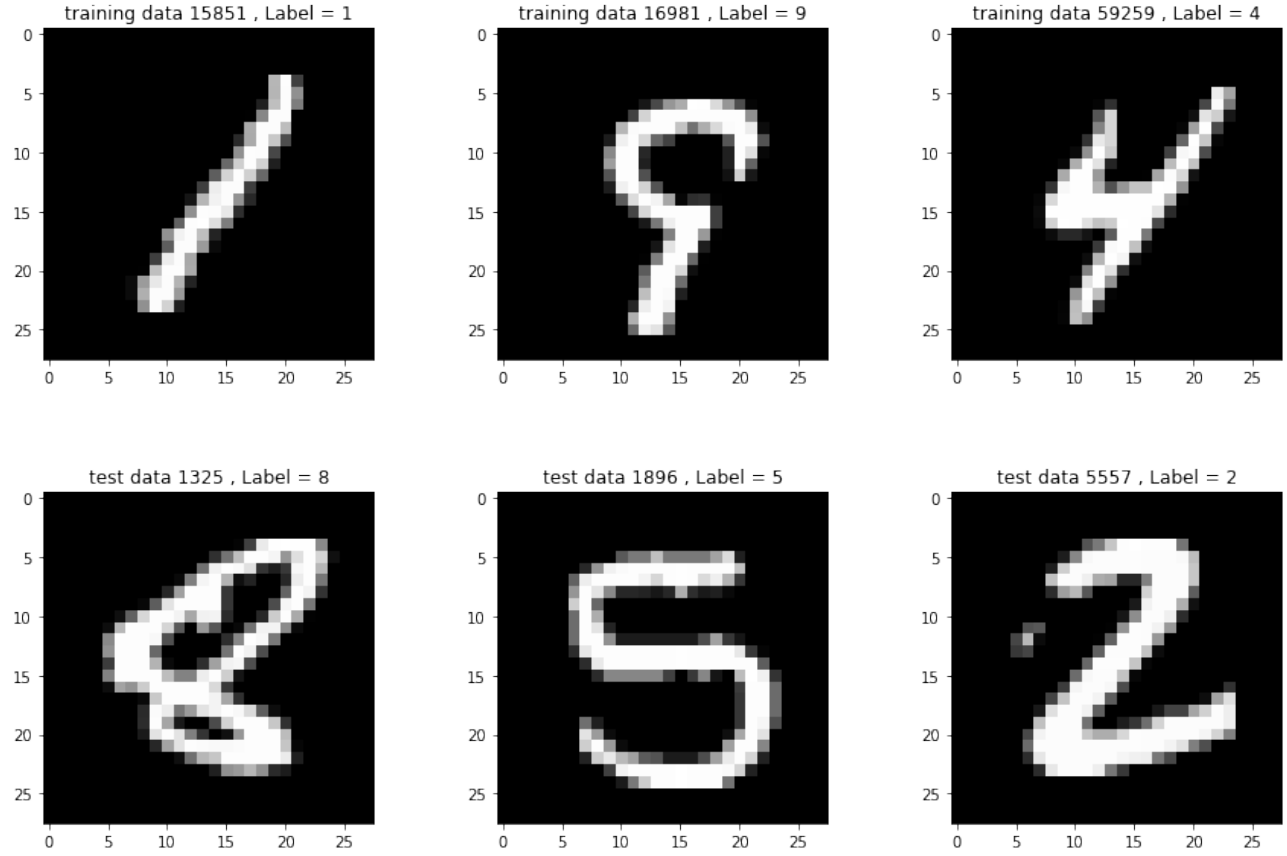
$$f'(z) = 1 - f^{2}(z)$$

(d). Implement and visualize them in the jupyter notebook. Discuss why gradient vanishing happens more when sigmoid-like activation functions are used.

You should see that for sigmoid-like activation functions (sigmoid and tanh), the derivatives are relatively small compared with ReLU. e.g. The maximum of derivative for sigmoid happens at $z = 0$, and it is only 0.25, whereas ReLU's derivative is constant 1 when $z \geq 0$. This makes sigmoid-like activation functions prone to the gradient vanishing problem.

3. Comparisons of different activation units on MNIST

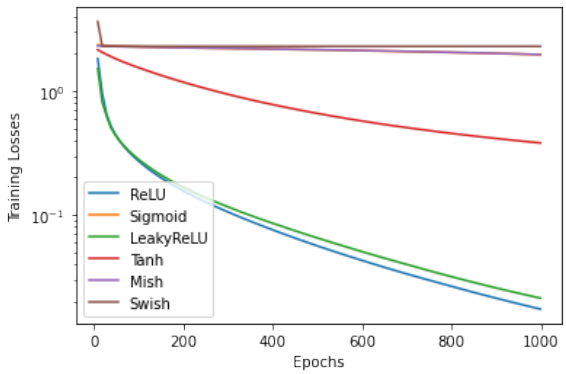
In previous problems, you should have learn that why we are going to use activation function in our models, and seen several kinds of activation function and derive their derivatives.
In this problem, we want to let you explore the performance of different activation units. The neural network architectures will be fixed, with only change of activation units. You will see that for different activation units, the convenge rate is different so we have to use different learning rate to optimize the training. And the resulting performance may also differ.
The dataset we use is MNIST, a renown dataset consists of hand-written digits images of 28*28 pixels. Since MNIST can be easily classified to over 99% accuracy, to better visualize the results, we use only a subset of it. Specifically, we use only the first 12000 training images from the dataset.
Please follow the instruction in the jupyter notebook and answer the question.
(a). Randomly visualize 3 training images and 3 test images. And for each image, specify the label.
Please see the sample code in the corresponding solution notebook

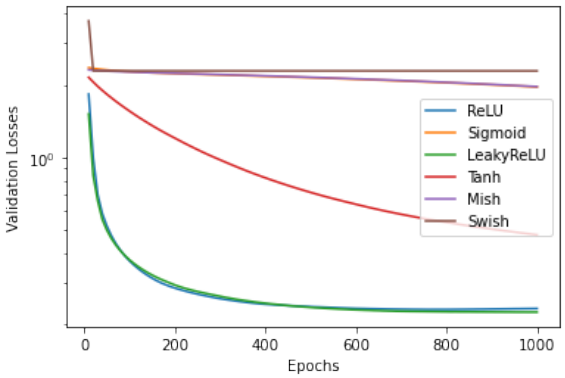


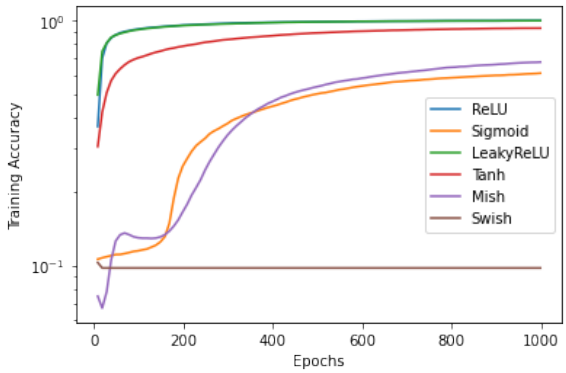
(b). Split the training data and validation data. You should use the 0.2 (in total 60000 images, 0.2 * 60000 = 12000 for our use) of the data for training and validation. The validation data should be the last 0.2 fraction of the training data. Implement the corresponding cells to complete the data preparation. The resulting number of training images should be 9600, and number of validation images should be 2400.
Please see the sample code in the corresponding solution notebook
(c). Implement the Mish and Swish activation function.
Please see the sample code in the corresponding solution notebook.
Note that
$\frac{\partial \sigma_{swish}(x)}{\partial x} = \frac{1}{1+e^{\beta x}} + x \cdot \frac{1}{1+e^{\beta x}} \cdot (1 - \frac{1}{1+e^{\beta x}}) \cdot \beta$
$\frac{\partial \sigma_{mish}(x)}{\partial x} = \tanh(\ln(1+e^x)) + x \cdot (1 - \tanh^2(\ln(1+e^x))) \cdot \frac{e^x}{1+e^x}$

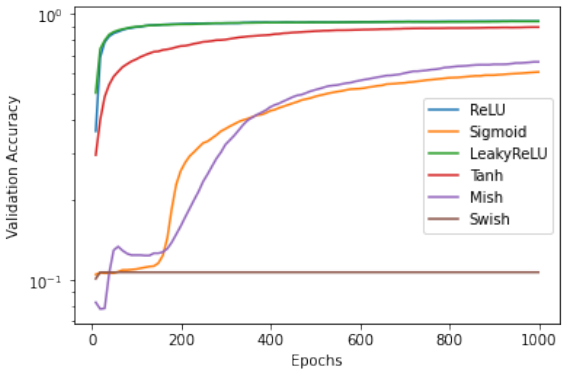
(d). Implement a neural network with 2 hidden fully-connected layers. The first layer (closer to the input)should have 200 neurons and the second layer should have 64 neurons. Immediately after both layers are the activation units. Train the neural networks with ReLU, Sigmoid, LeakyReLU, Tanh, Mish, and Swish using learning rate 0.005 for 1000 epochs. Visualize the training loss, validation loss, training accuracy, and validation accuracy along epochs for different activation units. What do you observe? Does the training converge? You should have 4 plots and answer the observation question.

We can see that the ReLU and LeakyReLU have similar loss during training, and their loss have converged. Their performance are the best compared with other activation functions. For Tanh, sigmoid, and Mish activation units, the loss do decrease but have not converged, which indicate that we should slightly increase the learning rate or increase the number of epochs. For Swish activation unit, the loss does not decrease and the accuracy remains low as that of random guess. This means either the model parameter is unsuitable for this activation function. Problems might include weight initialization, optimization parameters. In the next problem we will see that this is actually caused by a high learning rate. By reducing the learning rate, the activation unit can work.





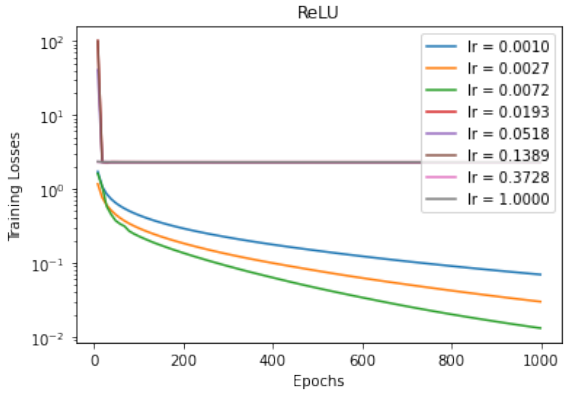


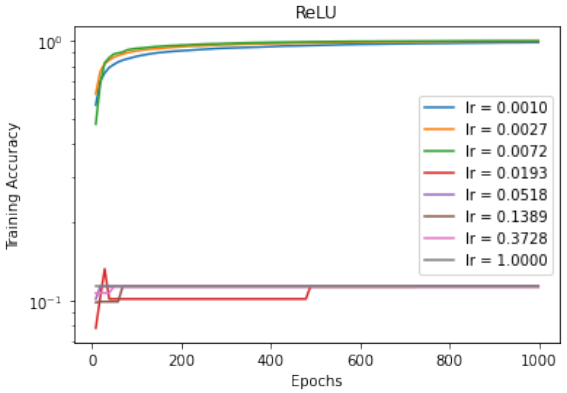


(e). Using the same network architecture, for each activation units, search for the learning rate that can make the training converges in 1000 epochs. What is the best learning rate for each activation units? Plot the comparison between different learning rates. You should have 24 plots in this question (training loss, validation loss, training accuracy, and validation accuracy for each activation units).

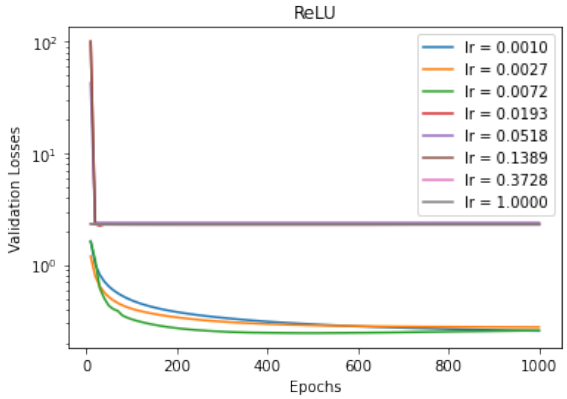
Using the validation set, the best learning rates for each activation unit are as follows:
ReLU: 0.001-0.0072,
Sigmoid: 1.0000,
LeakyReLU: 0.0027-0.0072,
Tanh: 0.1389-0.3728,
Mish: 0.3728-1.0000,
Swish: 0.001-0.0027,

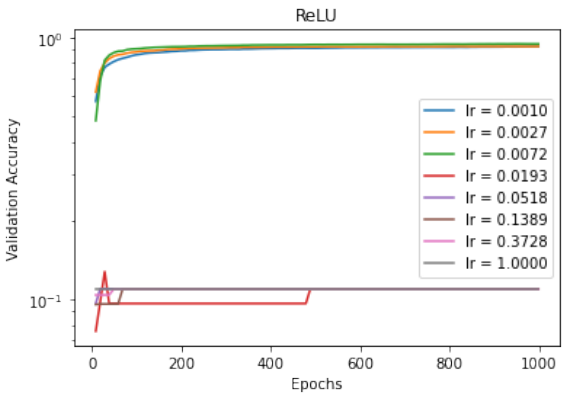
Training of ReLU



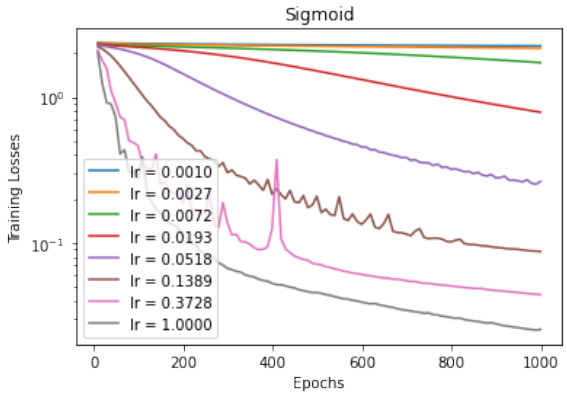


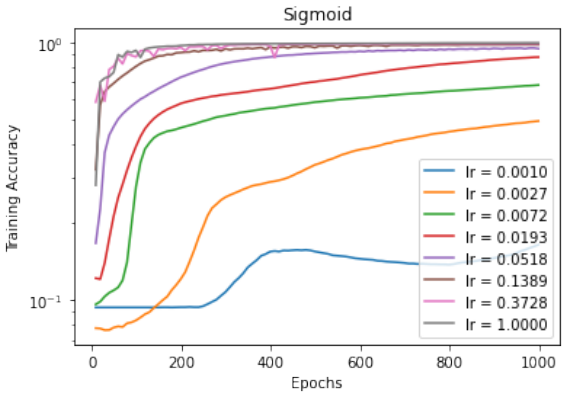
Validation of ReLU



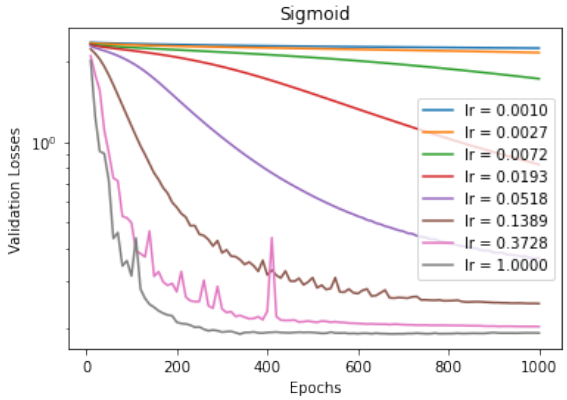


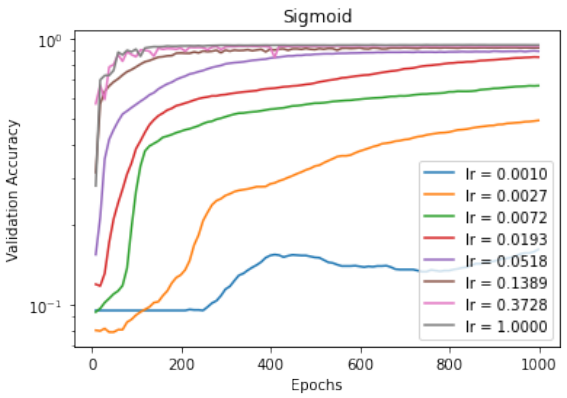
Training of Sigmoid



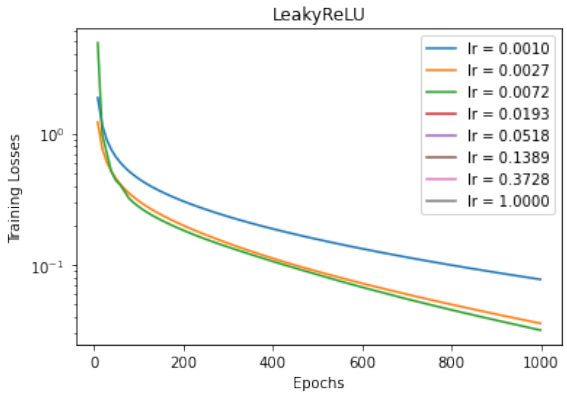


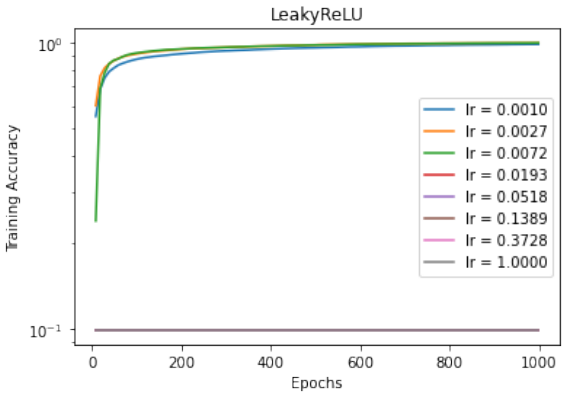
Validation of Sigmoid



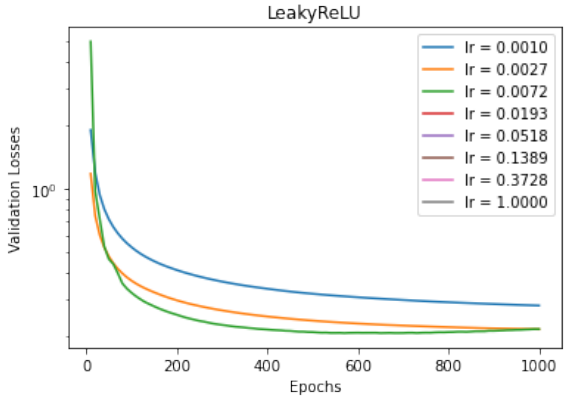


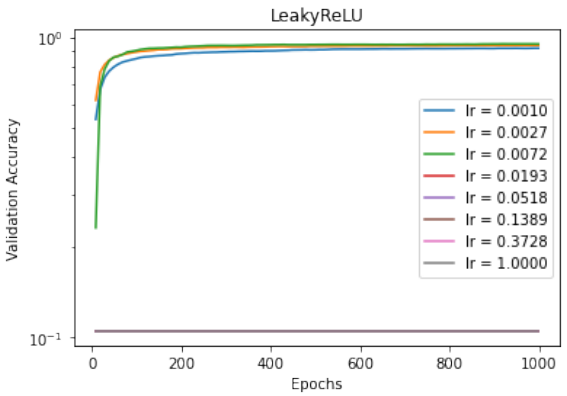
Training of LeakyReLU



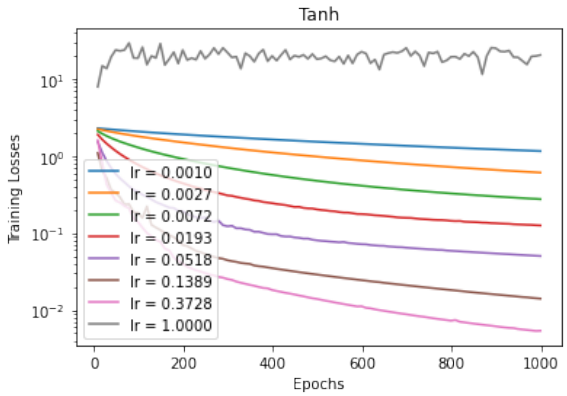


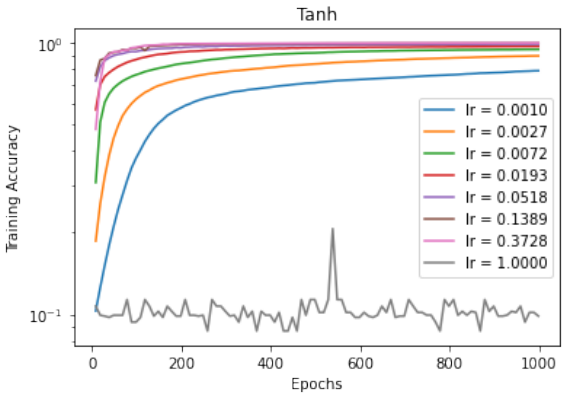
Validation of LeakyReLU



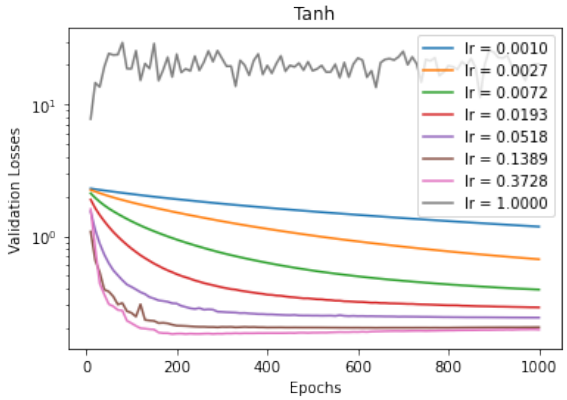


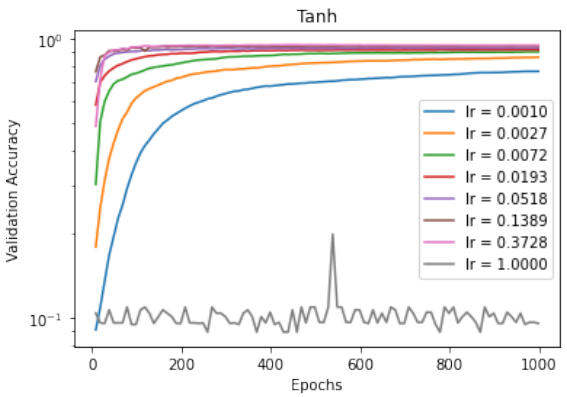
Training of Tanh



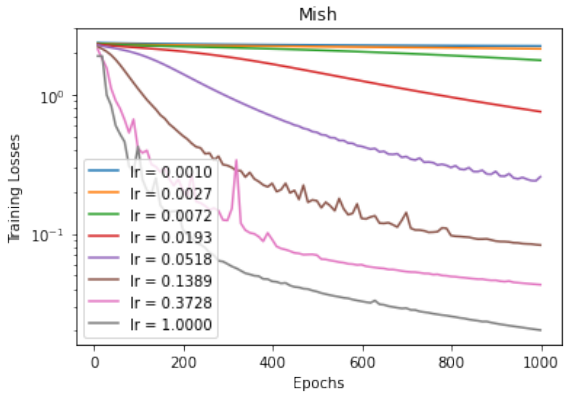


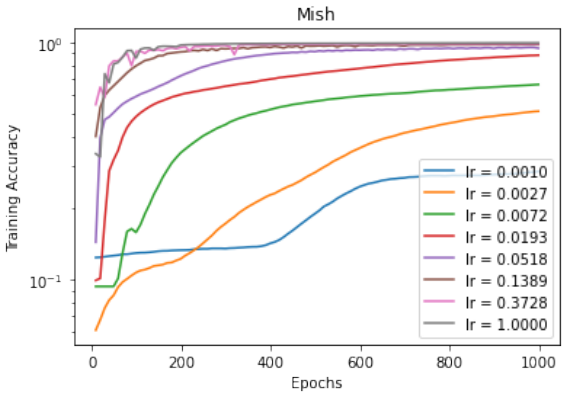
Validation of Tanh



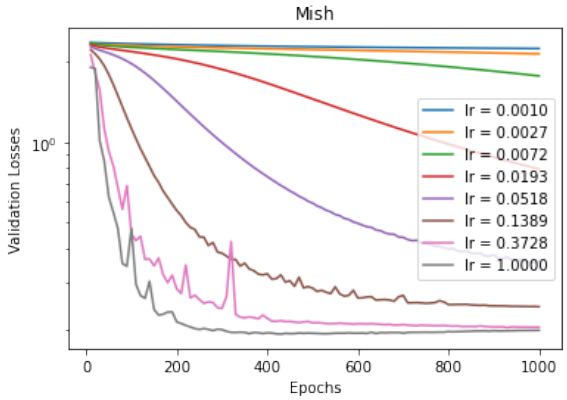


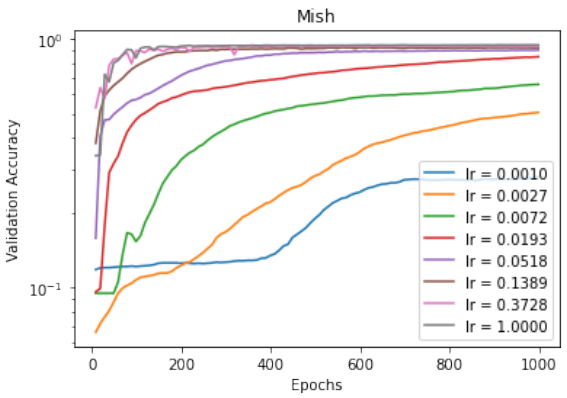
Training of Mish



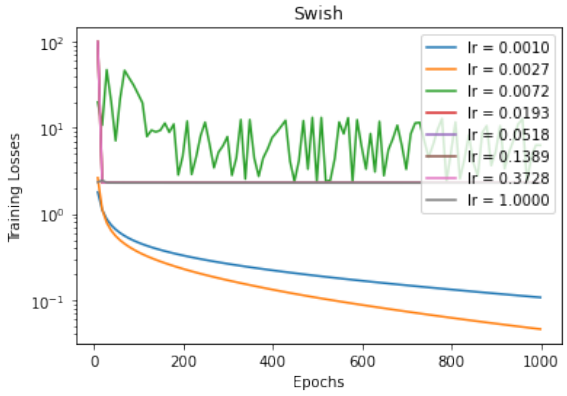


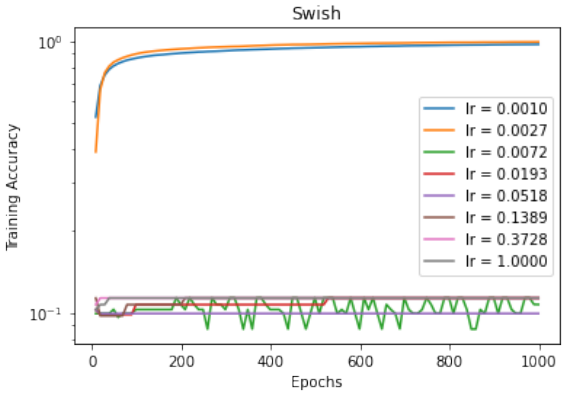
Validation of Mish



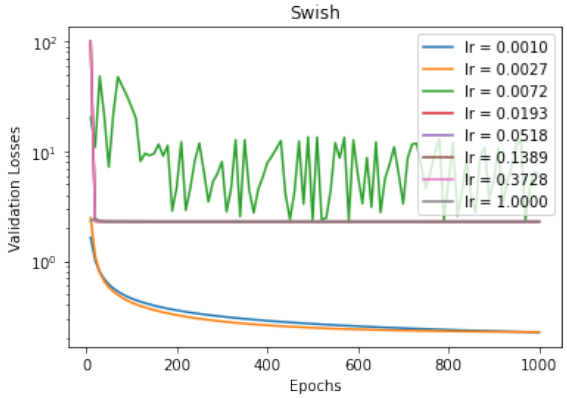


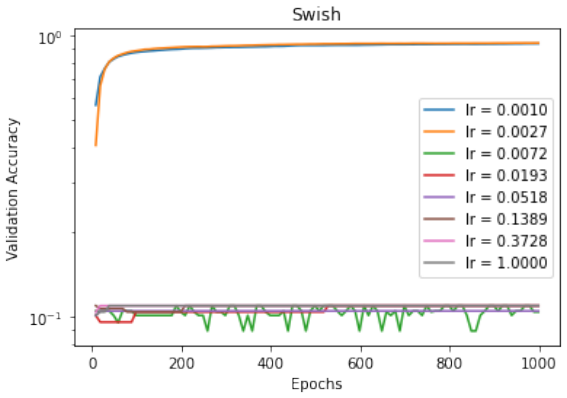
Training of Swish





Validation of Swish





(f). Using the best learning rate for each activation units obtained from the previous question, train the network again **using both training data and validation data**. What is the classification accuracy for each activation unit?

You might get different accuracy but the accuracy of all activation units should be larger than 0.92.

Accuracy of using ReLU activation unit at learning rate 0.0027: 0.9293
Accuracy of using Sigmoid activation unit at learning rate 1.0000: 0.9427
Accuracy of using LeakyReLU activation unit at learning rate 0.0027: 0.9305
Accuracy of using Tanh activation unit at learning rate 0.3728: 0.9451
Accuracy of using Mish activation unit at learning rate 0.3728: 0.9343
Accuracy of using Swish activation unit at learning rate 0.0010: 0.9229

4. Initialization and Activation Units: A Numerical Stability's Perspective

We implement our neural network models with concrete machines with limited precisions. It is essential to consider numerical stability. In this problem, we will show you that initialization of parameters is quite important due to numerical stability issues. Moreover, we will explore initialization policies for different activation units. Although initialization and activation units are orthogonal concepts, in practice they are often entangled. It is very important to choose initialization methods according to your activation units in neural networks!

In the coding part of this problem, we will use PyTorch, a deep learning framework with automatic differentiation, intensively. We believe that it will save you from the tedious work of calculating the gradients and keeping pluging them into the codes manually. Besides, you should be exposed to such modern frameworks for engineering practiceas early as possible although using frameworks should be taught later according to the course schedule.

(a). This part aims to help you review some basic issues of numerical methods in computer science, which is the foundation of the whole problem.

Please check the jupyter notebook and **answer questions** following the instructions.

Solution Here

(b). Consider a neural network (MLP) with L layers in total, the input is $\mathbf{x} \in \mathbb{R}^{n^{(0)}}$, the output is $\mathbf{y} \in \mathbb{R}^{n^{(L)}}$, the activation function is $\sigma : \mathbb{R} \to \mathbb{R}$ (we use the same activation function for all hidden layers), and the weight for layer l is $\mathbf{W}^{(l)}$, where $l \in \{1, 2, \ldots, L\}$. For part (b), we ignore the bias $\mathbf{b}^{(l)}$ in each layer for simplicity, since we can augment the input $\mathbf{x}$ with a constant entry 1 to achieve the same goal.
For forward propagation, we have:

$$\mathbf{y} = \sigma(\mathbf{W}^{(L)}\ \sigma(\mathbf{W}^{(L-1)}\ ...\ \sigma(\mathbf{W}^{(1)}\ \mathbf{x})...))$$

$$\mathbf{W}^{(l)} \in \mathbb{R}^{n^{(l)} \times n^{(l-1)}}$$

For convenience, we introduce some notations for intermediate variables. Let

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)},\ l > 0$$
$$\mathbf{a}^{(l)} = \begin{cases} \sigma(\mathbf{z}^{(l)}),\ l > 0 \\ \mathbf{x},\ l = 0 \end{cases}$$

Then the forward propagation algorithm can be rewritten as:

$$\begin{aligned} &\mathbf{a}^{(0)} = \mathbf{x} \\ &\text{for } l \leftarrow 1 \text{ to } L \text{ do} \\ &\quad \mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)},\ \mathbf{a}^{(l)} = \sigma(\mathbf{z}^{(l)}) \\ &\mathbf{y} = \mathbf{a}^{(L)} \end{aligned}$$

Assume that the final loss is $\mathcal{L}$ and that we know the derivative of loss w.r.t the output $\mathbf{y}$, i.e. $\dfrac{\partial \mathcal{L}}{\partial \mathbf{y}^T}$. Then the back-propagation algorithm can be write as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(L)T}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}^T}$$

for $l \leftarrow L$ downto 1 do

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)T}} = \sigma'(\mathbf{z}^{(l)}) \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(l)T}},\ \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(l-1)T}} = \mathbf{W}^{(l)T} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)T}},\ \nabla_{\mathbf{W}^{(l)}} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)T}} \mathbf{a}^{(l-1)T}$$

Let

$$\begin{aligned}\delta^{(l)} &= \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)T}} \\ &= \mathrm{diag}(\sigma'(\mathbf{z}^{(l)})) \mathbf{W}^{(l+1)T} \mathrm{diag}(\sigma'(\mathbf{z}^{(l+1)})) \cdots \mathbf{W}^{(L)T} \mathrm{diag}(\sigma'(\mathbf{z}^{(L)})) \frac{\partial \mathcal{L}}{\partial \mathbf{y}^T}\end{aligned}$$

Note that $\mathrm{diag}(\sigma'(\mathbf{z}^{(l)}))$ is a diagonal matrix consists of the derivative of $\mathbf{a}^{(l)}$ w.r.t $\mathbf{z}^{(l)}$ (i.e., $\frac{\partial \sigma(\mathbf{a}^{(l)})}{\partial \mathbf{z}^{(l)}}$).

We can write the gradient of loss w.r.t the parameter $\mathbf{W}^{(l)}$, $\nabla_{\mathbf{W}^{(l)}} \mathcal{L}$ as:

$$\nabla_{\mathbf{W}^{(l)}} \mathcal{L} = \delta^{(l)} \mathbf{a}^{(l-1)T}$$

In summary, the gradient $\nabla_{\mathbf{W}^{(l)}} \mathcal{L}$ is the product of many matrices. The computation procedure is prone to numerical stability issues, especially for deep neural networks where L is quite large. We can take these matrices as random with i.i.d. elements. Rigorous theory about random matrices is beyond the scope of this course, but you can imagine that if those diagonal matrices and weight matrices are interchangeable, you can estimate the gradient with the multiplication of $\Pi_l\ diag\{\sigma'(z_i^{(l)})\}$ and $\Pi_l\ \mathbf{W}^{(l)T}$. **The numerical values of $\mathbf{W}^{(l)}$ matrices and the characteristics of σ' are equally important, and they are usually coupled. We need to consider the initialization problem holistically.** In a real-world training procedure, we usually use the mean of $\nabla_{\mathbf{W}^{(l)}} \mathcal{L}$ over a batch of different samples. But this fact does not affect our analysis here because the summation of zeros/infinities/NaNs is still zero/infinity/NaN.

Suppose that all matrices/vectors in the equation (???) are all random matrices/vectors with i.i.d. elements, and each matrix/vector is independent of other matrices/vectors. To be more precise, we have following assumptions:

$$\forall l, \forall i \in \{1, ..., n^{(l-1)}\}, \sigma'(z_i^{(l)}) \sim F_{\sigma'^{(l)}}$$

$$\forall l, \forall i \in \{1, ..., n^{(l)}\}, \forall j \in \{1, ..., n^{(l-1)}\}, W_{ij}^{(l)} \sim F_{\mathbf{W}^{(l)}}$$

$$\forall i \in \{1, ..., n^{(L)}\}, \nabla_y L[i] \sim F_{\nabla_y L}$$

$$\forall l, \forall i \in \{1, ..., n^{(l)}\}, a_i^{(l)} \sim F_{\mathbf{a}^{(l)}}$$

Here, $F_{\sigma'^{(l)}}$, $F_{\mathbf{W}^{(l)}}$, $F_{\nabla_y L}$ and $F_{\mathbf{a}^{(l)}}$ are all independent scalar probability distributions. Given the expected values and variances of these distributions, that is $\mathbb{E}[F_{\sigma'^{(l)}}]$, $Var[F_{\sigma'^{(l)}}]$, $\mathbb{E}[F_{\mathbf{W}^{(l)}}]$, $Var[F_{\mathbf{W}^{(l)}}]$, $\mathbb{E}[F_{\nabla_y L}]$, $Var[F_{\nabla_y L}]$, $\mathbb{E}[F_{\mathbf{a}^{(l)}}]$, and $Var[F_{\mathbf{a}^{(l)}}]$, please **write the expected value and variance of the gradient $\nabla_{\mathbf{W}^{(l)}} L$ using these given values**. Of course, knowing more details, like what kind of function $\sigma(\cdot)$ is and the distribution that the input $\mathbf{x}$ follows, we may not need so many given values. In later parts, you will dive deeper into this topic. In this part, just let us assume that all these random variables are independent.

Solution Here

(c). In this part, we aim to give to give you some intuitions on numerical issues of random matrices albeit that the mathematics background of this topic is out of our hands. Also, you may use this part's codes to verify your answer in last part.
After reading the background information and finishing the formula derivation above, please **complete codes** in the jupyter notebook and **answer questions** following the instructions.

Solution Here

(d). In last part, we can control each random matrix independently. But in real neural networks, some of them are entangled in a non-linear way, making the analysis much more difficult. For example, the values of $\mathbb{E}[F_{\sigma'(l)}]$ are tightly encoupled with the characteristics of the function σ', which can be quite complex. Putting all these complex components together, neural networks with non-linear activation units are very sensitive numerically. There are many related topics. In this assignment problem, we only focus on the initialization story.

In the coding part, we will show that **even a very simple problem with a few layers, an inappropriate initialization can mess everything up**. At the first glance, you might think how could things go wrong in our demo? But unfortunately, even a slight mistake in parameter initialization can cause unexpected catastrophe in neural networks with non-linear activation units. We aim to give you a sense that those non-linear activation units are like geniuses with bad tempoer. They can be as terrible as powerful. s

Similar to gradient vanishing, we may also have the gradient exploding issue. Instead of using sigmoid, in this part we use ReLU to demonstrate this phenomenon because sigmoid is less prone to gradient exploding.

Please **complete codes** in the jupyter notebook and **answer questions** following the instructions.

Solution Here

(e). After realizing the issue, researchers are working hard to solve/mitigate it in recent years. In this part, we will introdcue **Xavier initialization** to solve the issue. In this part, we will still use the same random matrix assumption as previous parts. $W_{ij}^{(l)}$, the i-th row, j-th column element of $\mathbf{W}^{(l)}$, is drawn from the same distribution $F_{\mathbf{W}^{(l)}}$ independently (not necessarily Gaussian). Moreover, we can assume that $\mathbf{E}[F_{\mathbf{W}^{(l)}}] = 0$ (think about why). $a_i^{(l)}$, the i-th entry of the activation vector $\mathbf{a}^{(l)}$, is drawn from the same distribution $F_{\mathbf{a}^{(l)}}$ independently (not necessarily Gaussian). For Xavier initialization, we **add the following assumptions** (clearly, if these assumptions cannot be satisfied, Xavier initialization shouldn't work well):

- $\sigma(-x) = -\sigma(x)$; the activation unit $\sigma(\cdot)$ is symmetric about 0.
- $\sigma'(x)|_{x=0} = 1$; the activation unit $\sigma(\cdot)$ can be approximated by $\mathbb{I}(x) = x$ near 0.

Now, imagine the gradient plots of common non-linear activation units like sigmoid, what is the "sweet spot"? Here, without justification, we stick to the following rules of thumb/ basic principles to initialize the weights appropriately (why?):

- $\mathbb{E}[F_{\mathbf{a}^{(l)}}] = 0$; the mean of the activations, should be zero.
- $Var[F_{\mathbf{a}^{(l)}}] = Var[F_{\mathbf{a}^{(l-1)}}]$; the variance of the activations should stay the same across every layer except for the first and last layer.

Please **prove that given the above principles and assumptions, the variance of $\mathbf{W}^{(l)}$ (i.e., $Var[F_{\mathbf{W}^{(l)}}]$) should satisify the following relationship** for each layer (except the first and the last layer, think about why) during the forward propagation:

$$n^{(l-1)}\ Var[F_{\mathbf{W}^{(l)}}] = 1 \text{ or equivalently } Var[F_{\mathbf{W}^{(l)}}] = \frac{1}{n^{(l-1)}}$$

And similarly, we have the following relationship during the backward propagation (**you don't need to prove this equation**):

$$n^{(l)}\ Var[F_{\mathbf{W}^{(l)}}] = 1 \text{ or equivalently } Var[F_{\mathbf{W}^{(l)}}] = \frac{1}{n^{(l)}}$$

In practice, we can just use the harmonic mean of these 2 estimations to estimate the initial parameters:

$$\frac{n^{(l-1)} + n^{(l)}}{2}\ Var[F_{\mathbf{W}^{(l)}}] = 1 \text{ or equivalently } Var[F_{\mathbf{W}^{(l)}}] = \frac{2}{n^{(l)} + n^{(l-1)}}$$

Then, please **apply the above equation to normal distribution and uniform distribution**. You need to represent the distributions parameters with $n^{(l)}$ and $n^{(l-1)}$. For a normal distribution $\mathcal{N}(0, \gamma^2)$, please **write an equation** $\gamma = \cdots$ **using** $n^{(l)}$ **and** $n^{(l-1)}$. For a uniform distribution $\mathcal{U}(-d, +d)$, please **write an equation** $d = \cdots$ **using** $n^{(l)}$ **and** $n^{(l-1)}$.

Solution Here

(f). After finishing the theory part above, please **check the codes in the jupyter notebook** and **implement Xavier initialization** in the coding part.

Solution Here

(g). In previous parts, we assumed that the activation units are are symmetric about 0 and can be approximated by $\mathbb{I}(x) = x$ near 0. By scaling weights and adjusting bias terms of the next layer (i.e., using affine transformations for the output of this layer's activation unit), we can convert many functions to have the same properties as activations units used in previous parts. Please **give one example activations unit whose inputs that can be affine-transformed a function that satisfies the assumptions in Xavier initialization (We need a brief justification)**. We suggest that justify it by construct an affine transformation and prove that the transformed function is valid for Xavier initialization.

Solution Here

(h). However, not every activation unit can satisify the assumptions in Xavier initialization or can be affine-transformed to such functions like your example in last part. For example, ReLU cannot be affine-transformed to a symmetric function around zero (think about why). So we have to develop a different initialization method for ReLU (and also other similar activations units). Without loss of generality, in this part we only talk about one initialization method desgined for ReLU called **Kaiming Initialization**.

We will still use the same random matrix assumption as previous parts. $W_{ij}^{(l)}$, the i-th row, j-th column element of $\mathbf{W}^{(l)}$, is drawn from the same distribution $F_{\mathbf{W}^{(l)}}$ independently (not necessarily Gaussian). Moreover, we can assume that $\mathbf{E}[F_{\mathbf{W}^{(l)}}] = 0$. $a_i^{(l)}$, the i-th entry of the activation vector $\mathbf{a}^{(l)}$, is drawn from the same distribution $F_{\mathbf{a}^{(l)}}$ independently. Now, the activation units become ReLU, $\sigma(x) = max(0, x)$.

For Kaiming initialization, we will twisted the rules of thumb a bit as follows (notice the difference!):

- $F_{\mathbf{a}^{(l)}}(x) = F_{\mathbf{a}^{(l)}}(-x)$; $F_{\mathbf{a}^{(l)}}$ is symmetric about the y-axis $x = 0$
- $Var[F_{\mathbf{a}^{(l)}}] = Var[F_{\mathbf{a}^{(l-1)}}]$; the variance of the activations should stay the same across every layer except for the first and last layer.

Now, **prove that given the above principles and assumptions, the variance of $\mathbf{W}^{(l)}$ (i.e., $Var[F_{\mathbf{W}^{(l)}}]$) should satisify the following relationship** for each layer (except the first and the last layer, think about why) during the forward propagation:

$$\frac{1}{2}n^{(l-1)}\ Var[F_{\mathbf{W}^{(l)}}] = 1 \text{ or equivalently } Var[F_{\mathbf{W}^{(l)}}] = \frac{2}{n^{(l-1)}}$$

Similar to previous parts, in practice, we can use the following formula to estimate the initial parameters:

$$\frac{n^{(l-1)} + n^{(l)}}{4}\ Var[F_{\mathbf{W}^{(l)}}] = 1 \text{ or equivalently } Var[F_{\mathbf{W}^{(l)}}] = \frac{4}{n^{(l)} + n^{(l-1)}}$$

Then, please **apply the above equation to normal distribution and uniform distribution**. You need to represent the distributions parameters with $n^{(l)}$ and $n^{(l-1)}$. For a normal distribution $\mathcal{N}(0, \gamma^2)$, please **write an equation** $\gamma = \cdots$ **using** $n^{(l)}$ **and** $n^{(l-1)}$. For a uniform distribution $\mathcal{U}(-d, +d)$, please **write an equation** $d = \cdots$ **using** $n^{(l)}$ **and** $n^{(l-1)}$.

Solution Here

(i). After finishing the theory part above, please **check the codes in the jupyter notebook** and **implement Kaiming initialization** in the coding part.

Solution Here

(j). Now, let us put them all together! The codes in the notebook will train the same neural network with different activation units and initialization methods for the same task. Please **run the codes**, **compare the results** and **answer questions following the instructions** in the notebook.

Solution Here