Study of Noise Injection and Gradient Regularization Toward Better Generalization

barnrang

October 2019

Contents

1 Introduction

Since the upcoming of neural network, many forms of model are created for tackling many kinds of task such as regression and classification. With the non-linearity introduced in the neural network, it holds a large capacity for expressing a complex task such as image classification or many non-linear regression. However, the high capacity might lead to overfitting or failure to generalize in unseen data which should be control with the process called regularization. Before neural network, there are several ways to regularize the linear regression such as ridge or Least Absolute Shrinkage and Selection Operator (LASSO). Likewise, regularization in neural network can be performed in many ways like data augmentation, for example, reflecting, shearing or bending the image, to expose the model with more variety of data or directly manipulating the structure or parameters of the model such as weight decay or dropout (reference here!!!).

1.1 Objective

The purpose of this thesis is to compare the mean-squared error for a regression task or the accuracy for a classification task with respect to the test data of the dataset by several regularization method both existed and our purposed methods. Several papers of using gradients to regularize the model, neural network, have been purposed such as Double backpropagation which it penalizes the loss function by the gradient of the loss with respect to the input in order to suppress the change occurred by small change in the input. Even though this method was purposed since 1998, this method can perform pretty well in some dataset like MNIST as the experiment conducted by Dániel Varga Gradient Regularization Improves Accuracy of Discriminative Models. In the other hand, there are several papers observing the regularization by injecting noise into the input or a layer of neural network. Based on the Taylor expansion, we can derive the relationship between the noise injection and gradient regularization. Despite knowing the fact, researches comparing the noise injection

and gradient regularization are scarce. Therefore, this paper will be mainly discussed the evaluation of the result performed by 2 categories of method are gradient regularization and noise injection especially Gaussian's noise injection into input space.

1.2 Organization

The structure of this paper is shown as follows,

In chapter 2, the fundamental of machine learning and deep learning will be explained, especially the composition of the neural network which is the main theme of this paper. It includes feed-forward neural network, activation function, convolution neural network and back propagation.

In chapter 3, the basic ideas of regularization for classical linear regression and neural networks will be introduced.

In chapter 4, the concept of the large margin classifier especially SVM will be explained as the basic concept for the related work.

In chapter 5, the researches and works related to gradient regularization and noise injection will be introduced which the methods derived from these works will be used in the experiments.

In chapter 6, the neural networks that will be used in the experiment will be described. Furthermore, the datasets and its usage will be introduced.

In chapter 7, our purposed regularization methods and its derivation will be explained. Then, the experimental methods and their results will be shown.

In chapter 8, the results will be summarized then discussed for further improvement in the future.

1.3 Research ethics and compliance to the codes of conducting research

2 Convolution Deep Neural Network

2.1 Feed-Forward Neural Network

2.1.1 Neuron Node

The neuron node represents the linear operation of input vector $\boldsymbol{x} \in \mathbb{R}^h$ with weight vector and bias scalar $\boldsymbol{w} \in \mathbb{R}^h, b \in \mathbb{R}$ and then it is applied by an non-linear activation function $f : \mathbb{R} \to \mathbb{R}$ which the result is the output of the neuron node. It can be written as the following equation.

$$y = f(\boldsymbol{w}^\top \boldsymbol{x} + b)$$

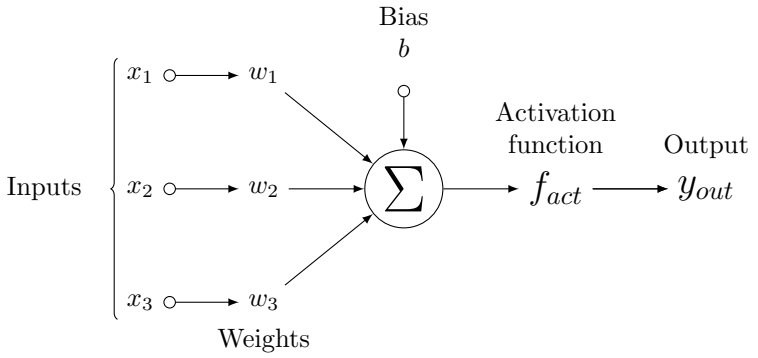


Figure 1: Diagram of a neuron node

2.1.2 Activation Functions

Biologically inspired, the activation function is represented as a gate which the incoming signal to a neuron node, in this case $\boldsymbol{w}^\top \boldsymbol{x} + b$, should be fired out to the next node or not. In the Neural Network context, the output values may vary infinitely on the range of $\mathbb{R}$, so that the activation function can determine the activated values that will be the inputs for the next neuron nodes. The usage and meaning might differ on the types of activation function.

The important property of the activation function is it needs to be non-linear function. It is important to introduce non-linearity into the neural network in order to increase the expressive power of it. In the other word, activation function helps the neural network to adapt better to arbitrary data.

Most of the activation functions likely to be differentiable so that the neural network can conduct backpropagation easily (will be mentioned in section 2.3). The most widely used nowadays are Tanh, sigmoid, Relu, LeakyRelu, or softmax and vice versa. The following items are the formulas for each activation function with respect to $x \in \mathbb{R}$

1. Tanh: $\dfrac{e^{2x} - 1}{e^{2x} + 1}$
2. Sigmoid: $\dfrac{1}{1 + e^{-x}}$
3. Relu (Rectified linear unit): $\max(0, x)$
4. Leaky Relu: $\begin{cases} x, & \text{if } x > 0 \\ ax, & \text{otherwise} \end{cases}$ (where a is a small constant)

For the softmax, it is employed to determine the probability of each element in $\boldsymbol{x} \in \mathbb{R}^h$. For any $\boldsymbol{x}$, it will constrain the sum of the element in the output vector to 1 which represent the probability of each element.

$$[\text{softmax}(\boldsymbol{x})]_i = \frac{\exp(x_i)}{\sum_{j=1}^{h} \exp(x_j)}$$

Note that $x_i \geq x_j \iff [\text{softmax}(\boldsymbol{x})]_i \geq [\text{softmax}(\boldsymbol{x})]_j$.

2.1.3 Neural Network

Trying to resemble the biological neural network system, neural network is a connection of many-to-many neuron node to pass the values through the layers of neuron. It can be illustrated as shown in Figure 2 which the neural network is composed of input nodes, one hidden layer nodes, and the output nodes.

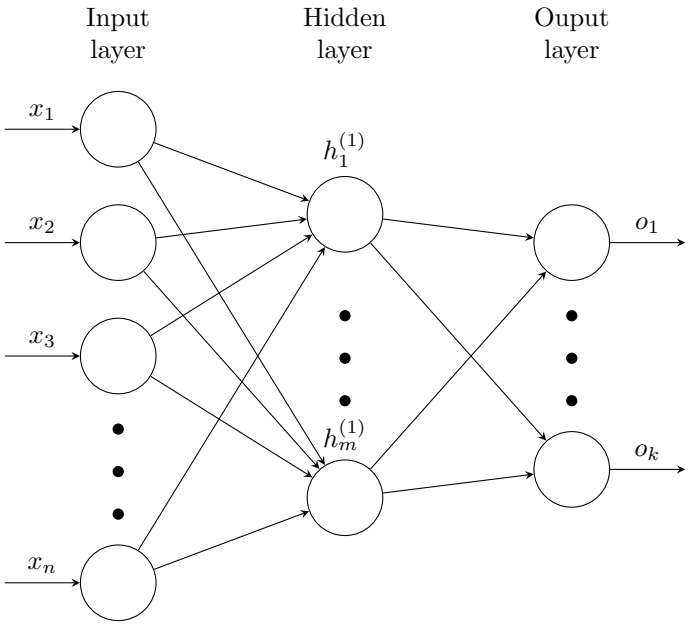


Figure 2: Neural network with one hidden layer

Similar to the neuron node, the computation of the neural network can be presented using the output from the previous layer $\boldsymbol{h}^{(i-1)} \in \mathbb{R}^m$ to the input for the next layer $\boldsymbol{h}^{(i)} \in \mathbb{R}^n$ by the weight matrix and bias vector $\boldsymbol{W}^{(i)} \in \mathbb{R}^{m \times n}, \boldsymbol{b}^{(i)} \in \mathbb{R}^n$ and activation function $f^{(i)}$ through the following equations.

$$\boldsymbol{h}^{(i)} = f({\boldsymbol{W}^{(i)}}^\top \boldsymbol{h}^{(i-1)} + \boldsymbol{b}^{(i)}) \tag{1}$$

For the neural network in Figure 2, the input layer is $\boldsymbol{h}^{(0)} = \boldsymbol{x}$ and the output layer is $\boldsymbol{o} = \boldsymbol{h}^{(2)}$

2.1.4 Loss Function

While training the neural network, we need to evaluate how well the neural network is performing in order to update the parameters, weights and biases, through backpropagation. The losses we will use in the experiment are Mean Squared Error (MSE), Binary-entropy loss and Cross-entropy loss. The formulas are as shown below with respect to the output $\boldsymbol{y} \in \mathbb{R}^n$ and the target $\hat{\boldsymbol{y}} \in \mathbb{R}^n$ where n is the number of samples.

1. MSE:

$$\mathcal{L}_{mse} = \frac{1}{n} \|\boldsymbol{y} - \hat{\boldsymbol{y}}\|_2^2 = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{2}$$

2. Binary-entropy loss:

$$\mathcal{L}_{binary} = \frac{1}{n} \sum_{i=1}^{n} (y_i (\log \hat{y}_i) + (1 - y_i)(\log(1 - \hat{y}_i))) \tag{3}$$

Cross-entropy loss is employed for multi-label classification. Let the output is $\boldsymbol{y} \in \mathbb{R}^{n \times c}$ and the target $\hat{\boldsymbol{y}} \in \mathbb{R}^{n \times c}$ where n, c are the number of samples and classes respectively. Here we will assume that the output was activated by softmax.

1. Cross-entropy loss:

$$\mathcal{L}_{cross} = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{c} \hat{y}_{ij} (\log y_{ij}) \tag{4}$$

It might be useful to note that the losses above are all convex with respect to the output.

2.2 Convolutional Neural Network

Convolutional networks (LeCun, 1989), or as known as Convolutional Neural Networks (CNNs) is a kind of network for processing data with grid-like topology. For instance, time-series data which can betaken a 1-D grid from each section of the data or, in this experiment, image data witch a group of 2-D grid pixel can be processed. The neuron in CNNs has a reception field adjusted by the window size and connects to other neuron overs the entire image. It can be interpreted that each neuron is detecting different features of the image, such as lines, curves or recognizing more complex features such as faces, texts and vice versa.

The very first CNNs by Y. LeCun as can be seen in figure 3. It succeeded in recognizing handwritten zip code and, later on, with higher computational power, a deeper or larger architectures were employed for many kind of image recognition tasks.

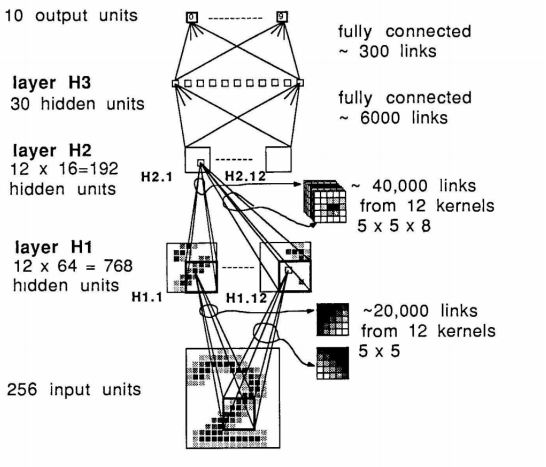


Figure 3: CNNs from the paper "Backpropagation Applied to Handwritten Zip Code Recognition"

2.2.1 Convolutional Layer

Convolutional layers are the main component of the CNNs which can be interpreted as feature extraction layers similarly to its original application in computer vision. In image processing, the convolution flips the kernel then conduct dot product on the reception field. The reason that the convolution need to be flipped is to conserve the commutative properties. Let I is the image and K is the kernel, we can write the convolution as the following

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$
$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n K(m,n)I(i-m, j-n)$$

However, in neural network the commutative properties are unnecessary which many library or implementation performs the **cross-correlation** instead

$$S(i,j) = (K \otimes I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

In CNNs, each layer with input sized $H \times W \times C$ consists the D kernels sized $h \times w \times C$ and bias sized D for each output channel will output a tensor with size $H_2 \times W_2 \times D$ which can be determined by the following formula

$$W_2 = \frac{W - w + 2P}{S} + 1$$
$$H_2 = \frac{H - h + 2P}{S} + 1$$

where P is padding size, S is stride value.

Mathematically, For a kernel $\boldsymbol{W} \in \mathbb{R}^{h \times w \times C_2 \times C_1}$, bias $\boldsymbol{b} \in \mathbb{R}^{C_2}$ and padded input $\boldsymbol{x}' \in \mathbb{R}^{H \times W \times C_1}$ which the output $\boldsymbol{y} \in \mathbb{R}^{H' \times W' \times C_2}$ where s is the stride, we can formulate the convolution as the following equation

$$y_{ijc} = \sum_{k=1}^{h} \sum_{l=1}^{w} \boldsymbol{w}_{klc}^{\top} \boldsymbol{x}'_{si+k-1, sj+l-1} + b_c \tag{5}$$

and the equation actually can be understood intuitively as figure 4. After the convolutional layer, it is usually followed by an activation function such as ReLU.

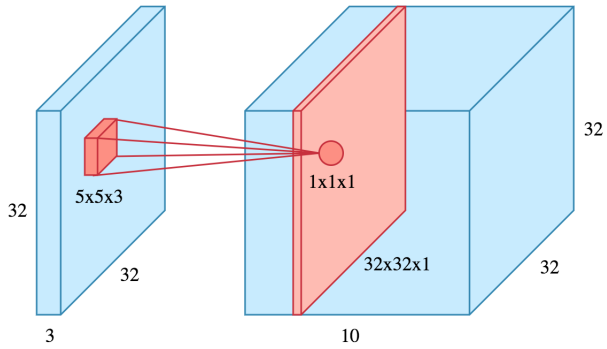


Figure 4: Illustration of a convolutional layer

2.2.2 Pooling Layer

After the convolutional layer and the non-linear activation function, we introduce further a layer to modify the the output called **pooling**. A pooling layer replaces the output with the summary statistics of the nearby output. For example, replace the 2×2 output field with its maximum element, max pooling, or with the average of all 4 elements, average pooling. Similar to convolutional layer, in this layer, mainly 2 parameters to control are the window size and the stride which the most usual combination is window size 2×2 and stride 2 as suggested in the figure 5.

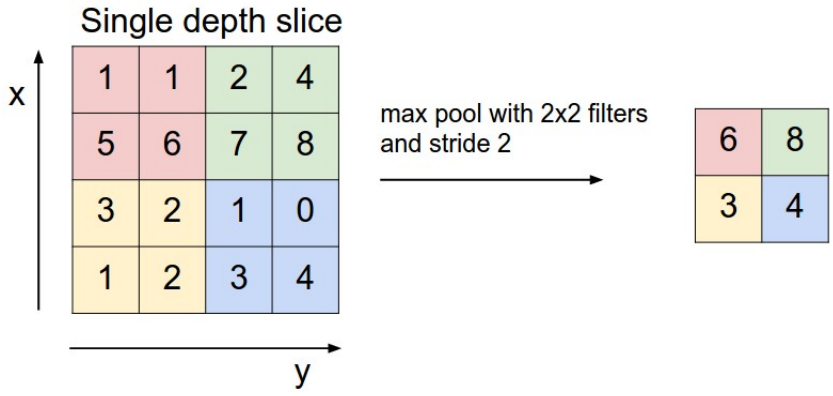


Figure 5: 2×2 max pooling with 2 stride

2.3 Backpropagation

In a simple linear regression such as $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b$ where the loss function is $\mathcal{L}(\boldsymbol{x}, y) = (f(\boldsymbol{x}) - y)^2$, the differentials of the loss with respect to the variable $\boldsymbol{w}, b$ are easily calculated by the formulas

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 2(f(\boldsymbol{x}) - y)\boldsymbol{x}$$
$$\frac{\partial \mathcal{L}}{\partial b} = 2(f(\boldsymbol{x}) - y)$$

which can be applied to optimization method such as gradient descent.

However, this is not the case for a neural network where there are a lot of layers stacking on each other and can be represented as the following equation

$$f(\boldsymbol{x}, \theta) = f_n(f_{n-1}(\cdots f_2(f_1(\boldsymbol{x}, \theta_1), \theta_2) \cdots, \theta_{n-1}), \theta_n)$$

and most of the layers are non-linear which the formula cannot be derived analytically. To alleviate this problem, rather than deriving the formula, instead exploiting the chain rule property is convenient especially for a layer-like structure such as neural network. The methodology will be explained through the computational graph and, later on, focus on each type of layer in the neural network.

2.3.1 Computational Graph

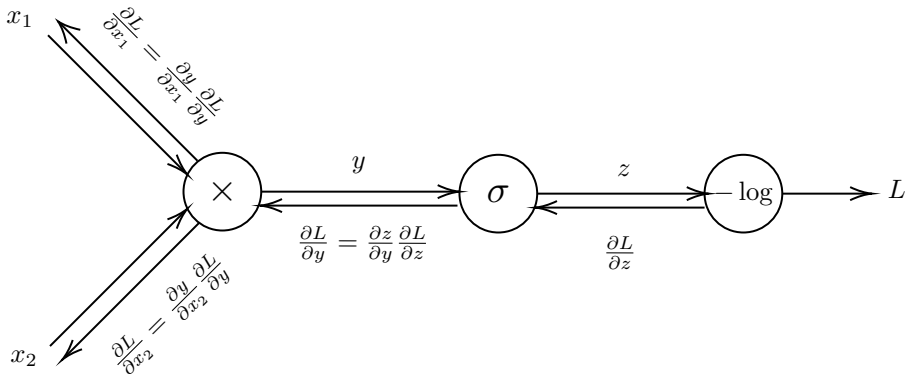


Figure 6: Simple Computational Graph

The computational graph represents the flow of calculation. As can be seen in the figure 6, the nodes represent the operation and the arrows from left to right represent the forward calculation, or forward pass, while the opposite direction arrows are what will be explained later, backpropagation. The equation of the sample computational graph can be written as follow,

$$
\begin{aligned}
y &= x_1 x_2 \\
z &= \frac{1}{1+\exp(-y)} \\
L &= -\log(z)
\end{aligned}
$$

In order to calculate $\partial\mathcal{L}/\partial x_1$ and $\partial\mathcal{L}/\partial x_2$, backpropagation can exploit the chain rule rather than directly derive a complex equation. This method might compute the differentiation, or gradient, with respect to variable which the complexity be as same as the forward pass. The derivation of the backward pass can be aligned as the following equations.

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial z} &= -\frac{1}{z} \\
\frac{\partial \mathcal{L}}{\partial y} &= \frac{\partial z}{\partial y}\frac{\partial \mathcal{L}}{\partial z} = z(1-z)\left(-\frac{1}{z}\right) = z-1 \\
\frac{\partial \mathcal{L}}{\partial x_1} &= \frac{\partial y}{\partial x_1}\frac{\partial \mathcal{L}}{\partial y} = x_2(z-1) \\
\frac{\partial \mathcal{L}}{\partial x_2} &= \frac{\partial y}{\partial x_2}\frac{\partial \mathcal{L}}{\partial y} = x_1(z-1)
\end{aligned}
$$

It can be observed that the inputs and outputs should be stored in order to compute the derivation for each step. Therefore, this method can be memory consuming for a larger neural network.

2.3.2 Backpropagation for Feed Forward Layer

Considering a single feed forward layer as equation 1, Let assume that the activation function is ReLU, the feed forward layer can be written step-by-step by the following equations. $\boldsymbol{h}^{(i-1)} \in \mathbb{R}^m, \boldsymbol{h}^{(i)} \in \mathbb{R}^n, \boldsymbol{W}^{(i)} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{b}^{(i)} \in \mathbb{R}^n$

$$\begin{aligned} \boldsymbol{h}'^{(i)} &= \boldsymbol{W}^{(i)^\top} \boldsymbol{h}^{(i-1)} + \boldsymbol{b}^{(i)} \\ \boldsymbol{h}^{(i)} &= \mathrm{ReLU}(\boldsymbol{h}'^{(i)}) \end{aligned}$$

The differentiation of ReLU can be written case-separately,

$$\mathrm{ReLU}(x) = \max(0, x) \Rightarrow \frac{\partial \mathrm{ReLU}(x)}{\partial x} = (x > 0) := \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

though the differentiation at $x = 0$ is undefined, The experiment conducted in this thesis based on the library Tensorflow (need ref!!!!) which treat it as 0.

For the differentiation $\partial \mathcal{L}/\partial \boldsymbol{W}^{(i)}, \partial \mathcal{L}/\partial \boldsymbol{b}^{(i)}$ and $\partial \mathcal{L}/\partial \boldsymbol{h}^{(i-1)}$ can be calculated on matrix multiplication.

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(i)}} &= \left(\frac{\partial \boldsymbol{h}'^{(i)}}{\partial \boldsymbol{W}^{(i)}} \right)^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}'^{(i)}} \\ \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(i)}} &= \left(\frac{\partial \boldsymbol{h}'^{(i)}}{\partial \boldsymbol{b}^{(i)}} \right)^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}'^{(i)}} \\ \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(i-1)}} &= \left(\frac{\partial \boldsymbol{h}'^{(i)}}{\partial \boldsymbol{h}^{(i-1)}} \right)^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}'^{(i)}} \end{aligned}$$

which the term $\partial \boldsymbol{h}'^{(i)}/\partial \boldsymbol{W}^{(i)}$ becomes 3-D tensor and others term that can be simplified for efficient gradient calculation as the following,

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(i)}} &= \left(\boldsymbol{h}^{(i-1)} \right)^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}'^{(i)}} \\ \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(i)}} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}'^{(i)}} \\ \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(i-1)}} &= \left(\boldsymbol{W}^{(i)} \right)^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}'^{(i)}} \end{aligned}$$

where $\partial \boldsymbol{h}'^{(i)}/\partial \boldsymbol{b}^{(i)} = \boldsymbol{I}, \partial \boldsymbol{h}'^{(i)}/\partial \boldsymbol{h}^{(i-1)} = \boldsymbol{W}^{(i)}$ and $\partial \mathcal{L}/\partial \boldsymbol{h}'^{(i)}$ can be calculated from the gradient with respect to the output $\boldsymbol{h}^{(i)}$ as the differentiation of ReLU in equation 6.

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}'^{(i)}} = \left(\boldsymbol{h}'^{(i)} > 0\right) \odot \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(i)}}$$

2.3.3 Backpropagation for Convolution Layer

From equation 5, we can see that convolution can be written similarly to a feed forward layer which the differentiation also turned out to be similar. It might be easier to accumulate the differentiation with respect to each element.

$$\begin{aligned}
\frac{\partial y_{ijc}}{\partial \boldsymbol{x}'_{si+k-1,sj+l-1}} &= \boldsymbol{w}_{klc} \\
\frac{\partial y_{ijc}}{\partial \boldsymbol{w}_{klc}} &= \boldsymbol{x}'_{si+k-1,sj+l-1} \\
\frac{\partial y_{ijc}}{\partial b_c} &= 1
\end{aligned}$$

This operation can be furthermore vectorized by the operation **im2col**, an operation to unfold the input and kernel for conducting matrix multiplication. After the operation, only a simple matrix multiplication is needed which greatly enhances the performance especially parallel processor such as GPU.

$$\boldsymbol{y} = (\hat{\boldsymbol{W}})^\top \hat{\boldsymbol{x}}' + \boldsymbol{b}$$

The backpropagation can be firstly done on $\hat{\boldsymbol{W}}$ and later fold it back to $\boldsymbol{W}$ by the inverse operation **col2im**.

2.3.4 Optimization

Optimization is an algorithm to minimize or maximize a function with respect to the inputs or parameters. In the neural network, we will minimize the loss function $\mathcal{L}(\theta; \boldsymbol{x}, \boldsymbol{y})$ where θ is the trainable parameters for the neural network $f(\boldsymbol{x}, \theta)$. Due to the accessibility to the differentiation of the neural network, we usually apply first-order optimization or occasionally second-order. Some example of algorithm that will be introduced here are Stochastic Gradient Descent (SGD), SGD with momentum, Nesterov Accelerated Gradient, Adaptive gradient (AdaGrad), Root Mean Square Propagation (RMSprop), and ADAM. Note that all the algorithm here might converge into the local minimum instead of global minimum.

1. Steepest Descent Method is the simplest method using the gradient as the steepest direction to decrease the value of the function within the neighborhood. For any function $f(\boldsymbol{x})$, we can minimize using this method with $\boldsymbol{x}_0$ as an initial points by the following update rule.

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha \nabla f(\boldsymbol{x}_k) \tag{7}$$

where α is the learning rate.

2. Stochastic Gradient Descent (SGD) is a method based on Steepest Descent Method. We train the parameter θ by randomly selecting training samples $\boldsymbol{x}^{(t)} \sim \boldsymbol{X}$ with training labels $\boldsymbol{y}^{(t)} \sim \boldsymbol{Y}$ then apply the Steepest Descent Method with respect to the parameters.

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta \mathcal{L}_{(\boldsymbol{x}^{(t)}, \boldsymbol{y}^{(t)}) \sim (\boldsymbol{X}, \boldsymbol{Y})}(\theta_t; \boldsymbol{x}^{(t)}, \boldsymbol{y}^{(t)}) \tag{8}$$

For the readability, the symbol represented sampling training data $(\boldsymbol{x}^{(t)}, \boldsymbol{y}^{(t)}) \sim (\boldsymbol{X}, \boldsymbol{Y})$ will be omitted.

3. SGD with Momentum employs the concept of momentum to make the direction of gradient change smoothly which empirically proved that it increases the speed of convergence.

$$\boldsymbol{v}_{t+1} = \beta \boldsymbol{v}_t + (1 - \beta) \nabla_\theta \mathcal{L}(\theta_t; \boldsymbol{x}^{(t)}, \boldsymbol{y}^{(t)}) \tag{9}$$

$$\theta_{t+1} = \theta_t - \alpha \boldsymbol{v}_{t+1} \tag{10}$$

for some $0 < \beta < 1$

4. Adaptive Gradient Algorithm (AdaGrad) accumulates the sum of the squared gradient to adjust the step length. When the magnitude of the gradient becomes large, by dividing with the accumulated squared gradient, it will reduce the step size so that it is less punishing for mistaken direction.

$$\boldsymbol{\Sigma}_{t+1} = \boldsymbol{\Sigma}_t + (\nabla_\theta \mathcal{L}(\theta_t))^2, \tag{11}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha \nabla_\theta \mathcal{L}(\theta_t)}{\sqrt{\boldsymbol{\Sigma}_{t+1}} + \epsilon} \tag{12}$$

5. Root Mean Square Propagation (RMSprop) is similar to Adagrad, but introduces momentum update to the accumulated squared gradient.

$$\boldsymbol{\Sigma}_{t+1} = \beta \boldsymbol{\Sigma}_t + (1 - \beta)(\nabla_\theta \mathcal{L}(\theta_t))^2, \tag{13}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha \nabla_\theta \mathcal{L}(\theta_t)}{\sqrt{\boldsymbol{\Sigma}_{t+1}} + \epsilon} \tag{14}$$

6. ADAM optimizer is combining AdaGrad, and the concept of momentum update to determine the direction of the step. This method is one of the most popular optimization method for neural network.

$$\boldsymbol{m}_{t+1} = \beta_1 \boldsymbol{m}_t + (1-\beta_1)\nabla_\theta \mathcal{L}(\theta_t), \tag{15}$$

$$\boldsymbol{v}_{t+1} = \beta_2 \boldsymbol{v}_t + (1-\beta_2)(\nabla_\theta \mathcal{L}(\theta_t))^2, \tag{16}$$

$$\hat{\boldsymbol{m}}_{t+1} = \frac{\boldsymbol{m}_{t+1}}{(1-\beta_1^t)} \tag{17}$$

$$\hat{\boldsymbol{v}}_{t+1} = \frac{\boldsymbol{v}_{t+1}}{(1-\beta_2^t)} \tag{18}$$

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{\boldsymbol{m}}_{t+1}}{\sqrt{\hat{\boldsymbol{v}}_{t+1}} + \epsilon} \tag{19}$$

2.4 Batch Normalization

In order to alleviate the vanishing gradient problem, batch normalization scales the input by its empirical mean and variance. It was claimed that (need ref) batch normalization can reduce overfitting due to the usage of data variation and also a larger learning rate can be used without exploding the gradients. The normalization for a batch with m samples $\boldsymbol{x} \in \mathbb{R}^m$ is shown as the following equation.

$$\mu_b = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{20}$$

$$\sigma_b^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_b)^2 \tag{21}$$

$$\hat{x_i} = \frac{x_i - \mu_b}{\sqrt{\sigma_b^2 + \epsilon}} \tag{22}$$

$$y_i = \gamma \hat{x_i} + \beta \tag{23}$$

Where μ_b, σ_b are batch mean and variance, $\hat{x_i}$ is normalized value of x_i, ϵ is a small constant to prevent from numerical instability, λ and β are parameters for scaling the normalized input which both of them are trainable.

For the test time, we might not be able to retrieve the empirical mean and variance. While training, the running mean and running variance $\hat{mu}, \hat{sigma}$ should be adjust by using momentum update.

$$\hat{\mu} := m\hat{\mu} + (1-m)\mu_b \tag{24}$$

$$\hat{\sigma} := m\hat{\sigma} + (1-m)\sigma_b \tag{25}$$

3 Regularization

In machine learning, the model is trained in order to generalize well to the unseen data so it should be regulated by some rule to not let the model get

too "used to" the training data. In this section, the regularization for linear regression and neural network will be introduced.

3.1 Regularization in Linear Regression

Briefly, the linear regression is the problem to minimize the MSE from n-sample target $\boldsymbol{y} \in \mathbb{R}^n$ with respect to the n-samples d-dimensional inputs $\boldsymbol{X} \in \mathbb{R}^{n \times (d+1)}$ by adjusting the weight $\boldsymbol{w} \in \mathbb{R}^{d+1}$, which an extra dimension is for bias. The equation

$$\mathcal{L}(\boldsymbol{w}, b; \boldsymbol{X}, y) = \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|_2^2 \tag{26}$$

can be solved analytically or using gradient descent. However, directly optimizing the loss in equation 26 might lose generality due to the outliers or bad features. Usually, it can be considered that the model is more expressive when the magnitude of $\boldsymbol{w}$ become large. This idea leads to the following regularization methods, Least Absolute Shrinkage and Selection Operator (LASSO) and Ridge.

3.1.1 LASSO

LASSO or as known as $L1$-regularizer penalizes the weight by it $L1$-norm by a parameter $\lambda \in \mathbb{R}$ which control the penalty due to the regularization. Instead, We can minimize the following loss function.

$$\mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}, \lambda) = \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|_2^2 + \lambda \|\boldsymbol{w}\|_1 \tag{27}$$

The analytical solution is fairly complex so it will not be covered here.

(explain more about sparse weights)

3.1.2 Ridge

Similar to LASSO, Ridge regression penalize the $L2$-norm of the weight and lead us to the optimization of the following loss function.

$$\mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}, \lambda) = \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|_2^2 + \lambda \|\boldsymbol{w}\|_2^2 \tag{28}$$

which this loss function can be solved analytically as below

$$\hat{\boldsymbol{w}} = (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})\boldsymbol{X}^\top \boldsymbol{y} \tag{29}$$

(explain more about smooth)

3.2 Regularization in Neural Network

Neural network works well with non-linear data, for example, nonlinear regression or non-linearly separable classification, because of its high expressive power due to the non-linear activation layers. However, it might cause the overfitting to the train data because of its own expressive power. As a regularizer, two main methods will be introduced which are weight decay and dropout. Some other

method to reduce overfitting might not be count as regularization, for instance, data augmentation which tries to expose the model to more data modified in a feasible range.

3.2.1 Weight Decay

Let consider a layer with weight $\boldsymbol{w} \in \mathbb{R}^d$. It can be penalized the weight similarly to the ridge regression as the following loss function.

$$\mathcal{L}_{decay}(\theta; \boldsymbol{X}, \boldsymbol{y}) = \mathcal{L}(\theta; \boldsymbol{X}, \boldsymbol{y}) + \lambda \left\|\boldsymbol{w}\right\|_2^2 \tag{30}$$

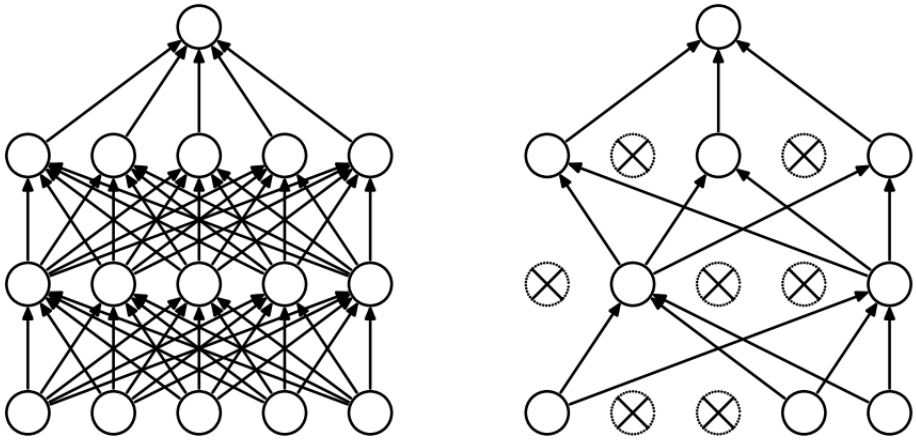
Assume that we apply SGD to update the parameter. Hence, the update with respect to the weight $\boldsymbol{w}$ will be

$$\begin{aligned} \boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \alpha \nabla_{\boldsymbol{w}} \mathcal{L} - 2\alpha\lambda \boldsymbol{w}_t & (31) \\ &= (1 - 2\alpha\lambda)\boldsymbol{w}_t - \alpha \nabla_{\boldsymbol{w}} \mathcal{L} & (32) \end{aligned}$$

which the weight $\boldsymbol{w}$ is shrinking proportionally. Therefore, this method is called "weight decay" or to reduce the magnitude of the weight so that it is less prone to overfitting problem. The factor λ can be adjust appropriately for each layer.

3.2.2 Dropout

Dropout is a method to drop some units, or values, to zero. It was claimed that (reference here!!!!) this method prevent the neural network from adapting or relying on a single node too much and encourage ensemble from averaging multiple units together when test time. Dropout can be illustrated as shown in figure 7



(a) Standard Neural Net (b) After applying dropout.

Figure 7: Dropout illustration

Consider a hidden layer l. Let $\boldsymbol{z}^{(l)}$ denote the vector of input into layer l, $\boldsymbol{y}^{(l)}$ denote the vector of output from layer l. Let $\boldsymbol{W}^{(l)}$ and $\boldsymbol{b}^{(l)}$ denote weight and bias of layer l, a standard feed-forward layer can be describe as

$$\begin{aligned} z_i^{l+1} &= \boldsymbol{w}_i^{l+1}\boldsymbol{y}^{(l)} + b_i^{(l)}, \\ y_i^{(l+1)} &= f(z_i^{(l+1)}) \end{aligned}$$

By applying dropout, a mask from Bernoulli distribution with probability p to be 1 will be generated.

$$\begin{aligned} r_j^{(l)} &\sim \text{Bernoulli}(p), \\ \widetilde{\boldsymbol{y}}^{(l)} &= \boldsymbol{r}^{(l)} \odot \boldsymbol{y}^{(l)}, \\ z_i^{l+1} &= \boldsymbol{w}_i^{l+1}\widetilde{\boldsymbol{y}}^{(l)} + b_i^{(l)}, \\ y_i^{(l+1)} &= f(z_i^{(l+1)}) \end{aligned}$$

4 Large Margin Classifier (SVM)

For simplicity, the classification problem to be discussed will return to two-class classification problem which can be written using linear models as the form

$$y(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}) + b \tag{33}$$

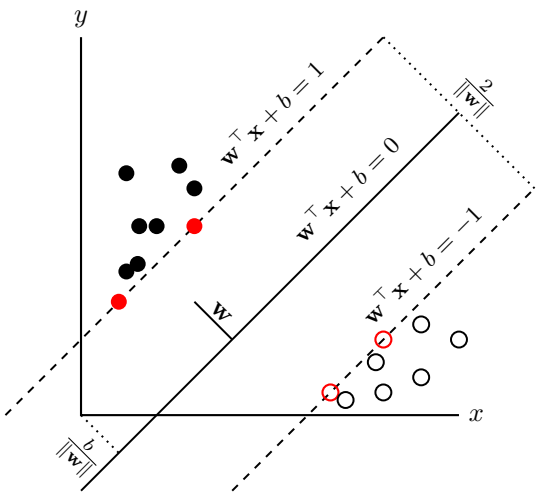


Figure 8: Decision boundary and its margin

where $\boldsymbol{\phi}(\boldsymbol{x})$ denotes a fixed feature-space transformation, such as Gaussian kernel $\exp(-x^2)$. The training dataset comprises N input vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ with their corresponding targets $t_1, \ldots, t_N$ where $t_n \in \{-1, 1\}$ and test data $\boldsymbol{x}$, new data points, will be classified based on the sign of $y(\boldsymbol{x})$.

In fact, there are many solutions to approach this problem that guarantee to fit the training data with finite iterations though the results might depend on the initial value of parameters $\boldsymbol{w}$ and b. To find the solution which generalizes the best, the support vector machine (SVM) tackles this problem through the concept of the margin, which is defined to be the smallest distance between the decision boundary and the samples as shown in figure 8. From Pattern Recognition textbook

The perpendicular distance of a arbitrary point $\boldsymbol{x}$ from a hyperplane $y(\boldsymbol{x}) = 0$ is given by $|y(\boldsymbol{x})| / \|\boldsymbol{w}\|$. If the case which all samples are classified correctly is considered, it can be easily shown that $t_n y(\boldsymbol{x}_n) > 0$. Therefore, the distance of a point $\boldsymbol{x}_n$ to the decision surface is

$$\frac{t_n y(\boldsymbol{x})}{\|\boldsymbol{w}\|} = \frac{t_n(\boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}) + b)}{\|\boldsymbol{w}\|}. \tag{34}$$

The margin is given by the smallest distance while we would like to maximize the margin. Thus the problem can be formulated as an optimization of $\boldsymbol{w}$ and b to maximize the distance

$$\underset{\boldsymbol{w}, b}{\operatorname{argmax}} \left\{ \frac{1}{\|\boldsymbol{w}\|} \min_n \left[t_n(\boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}_n) + b) \right] \right\} \tag{35}$$

However, directly solve equation 35 is complicated and further simplify can be done. From an observation, the distance from any point $\boldsymbol{x}_n$ is unchanged for

any rescaling $\boldsymbol{w} \to \tau\boldsymbol{w}$ and $b \to \tau b$ so it can be used to freely set

$$t_n(\boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}_n) + b) = 1 \tag{36}$$

for the point that is closest to the boundary and for all data points, they satisfy

$$t_n(\boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}_n) + b) \geq 1 \tag{37}$$

Furthermore, the optimization in equation 35 can be simplified into

$$\underset{\boldsymbol{w}}{\operatorname{argmax}} \frac{1}{\|\boldsymbol{w}\|} \iff \underset{\boldsymbol{w}}{\operatorname{argmin}} \frac{1}{2} \|\boldsymbol{w}\|^2 \tag{38}$$

where the factor $1/2$ is included for later convenience. The problem is turned into a quadratic programming problem which an equation is optimized while there is a set of inequality constraints. Even the variable b disappears from the optimization, it is implicitly adjusted by the constraints. In order to solve the problem, the Lagrange multipliers $a_n \geq 0$ are introduced giving Lagrangian function

$$L(\boldsymbol{w}, b, \boldsymbol{a}) = \frac{1}{2} \|\boldsymbol{w}\|^2 - \sum_{n=1}^{N} a_n \left\{ t_n(\boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}_n) + b) - 1 \right\} \tag{39}$$

By taking partial derivative with respect to $\boldsymbol{w}$ and b and let them be zeros, two conditions are obtained

$$\partial L/\partial \boldsymbol{w} = 0 \quad (\Rightarrow) \quad \boldsymbol{w} = \sum_{n=1}^{N} a_n t_n \boldsymbol{\phi}(\boldsymbol{x}_n) \tag{40}$$

$$\partial L/\partial b = 0 \quad (\Rightarrow) \quad 0 = \sum_{n=1}^{N} a_n t_n \tag{41}$$

Next, the term $\boldsymbol{w}$ in (39) can be eliminated by substituting $\boldsymbol{w}$ from (40) and b can be eliminated because its coefficient is exactly the same as RHS of (41). Therefore, we derive an equation relying only on $\boldsymbol{a}$ as called as **dual representation**

$$\tilde{L}(\boldsymbol{a}) = \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m t_n t_m \boldsymbol{\phi}(\boldsymbol{x}_n)^\top \boldsymbol{\phi}(\boldsymbol{x}_m) \tag{42}$$

with the following constraints

$$a_n \geq 0 \tag{43}$$

$$\sum_{n=1}^{N} a_n t_n = 0 \tag{44}$$

This problem, a_n, can be solved by quadratic programming which will be omitted. Instead of solving $\boldsymbol{w}$ and b, we solve a_n which can be used for deriving $\boldsymbol{w}$

and b in the later steps. Substitute $\boldsymbol{w}$ in (40) into (33), the linear model will be represented in the form of a_n and b as

$$y(\boldsymbol{x}) = \sum_{n=1}^{N} a_n t_n \boldsymbol{\phi}(\boldsymbol{x})^\top \boldsymbol{\phi}(\boldsymbol{x}_n) + b = \sum_{n=1}^{N} a_n t_n k(\boldsymbol{x}, \boldsymbol{x}_n) + b \tag{45}$$

where $k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\phi}(\boldsymbol{x})^\top \boldsymbol{\phi}(\boldsymbol{x}')$ is called **kernel function**. Here is the list of most used kernel functions.

Kernel name	Formula
Linear	$\boldsymbol{x}'^\top \boldsymbol{x}$
RBF (Gaussian)	$\exp(-\left\|\boldsymbol{x} - \boldsymbol{x}'\right\|^2 / 2\sigma^2)$
Polynomial	$(1 + \boldsymbol{x}^\top \boldsymbol{x})^d$

Table 1: List of typical kernel functions

From the Karush–Kuhn–Tucker conditions (KKT), the complementary slackness is introduced which the following condition should be satisfied.

$$a_n \{t_n y(\boldsymbol{x}_n) - 1\} = 0 \tag{46}$$

For any data point $\boldsymbol{x}_n$ which $t_n y(\boldsymbol{x}_n) - 1 > 0$, a_n should be zero. Therefore, $a_n \neq 0$ if and only if $t_n y(\boldsymbol{x}_n) - 1 = 0$ or only on the data points which lying on the maximum margin hyperplane. From this point, only data points that come into consideration are only a few points on the maximum margin hyperplane and the set S is defined as the indices of the points that are "active" and it is called **support vector**. Here, we can rearrange (45) as the following for less computational complexity

$$y(\boldsymbol{x}) = \sum_{n \in S} a_n t_n k(\boldsymbol{x}, \boldsymbol{x}_n) + b \tag{47}$$

For any support vector $\boldsymbol{x}_n$, it is satisfied $t_n y(\boldsymbol{x}_n) = 1$ then substitute by (45), we yield

$$t_n \left(\sum_{m \in S} a_m t_m k(\boldsymbol{x}_m, \boldsymbol{x}_n) + b \right) = 1 \tag{48}$$

To determine the value of variable b, it is possible to pick any support vector $\boldsymbol{x}_n$ and solve (48) directly. However, that might cause numerical instability; hence, instead, we multiply (48) by t_n, which $t_n^2 = 1$, and average them over all support vector then the answer for b is

$$b = \frac{1}{N_s} \sum_{n \in S} \left(t_n - \sum_{m \in S} a_m t_m k(\boldsymbol{x}_m, \boldsymbol{x}_n) \right) \tag{49}$$

where N_s is the number of support vector ($N_s = |S|$).

5 Related Work

As introduced in the previous sections, there are number of ways to regularize the neural network such as weight decay and dropout. Meanwhile, another idea of regularizing the gradient instead to suppress large change with respect to a small change in the input space or directly inject the noise to the input space or hidden layer.

5.1 Gradient Regularization

Gradient regularization is introduced by many intepretation, such as encouraging smoothness of the output space or decision boundary, or encouraging a large margin decision boundary which they will be introduced in this section. For convenience, we will define our model, neural network, and the loss as the following.

$$\mathcal{L}(\boldsymbol{x}, y, \theta) = M(f(\boldsymbol{x}, \theta), y) = M(\text{softmax}(g(\boldsymbol{x}, \theta), y)) \tag{50}$$

5.1.1 Double Backpropagation

This idea is the very first purposed one since 1991 by Y. LeCun. It penalizes the loss by the gradient of the loss with respect to the input as

$$\mathcal{L}_{DG}(\boldsymbol{x}, y; \theta) = \mathcal{L}(\boldsymbol{x}, y; \theta) + \lambda \left\| \partial \mathcal{L} / \partial \boldsymbol{x} \right\|_2^2 . \tag{51}$$

The loss function explicitly pushes the norm of the gradient to zeros which encourage the minimum to be broader and increases generalization. It has been proved to be succeed on a handwritten numbers dataset.

5.1.2 Jacobian Regularization

1. **Jacobian Regularizer (JacReg)** Penalizes the squared Frobenius norm of the Jacobian of the softmax output with respect to the input

   $$\mathcal{L}_{JacReg}(\boldsymbol{x}, y; \theta) = \mathcal{L}(\boldsymbol{x}, y; \theta) + \lambda \left\| J_f \right\|_F^2 \tag{52}$$

   This work was done by J. Sokolic 2017. Inspired by large margin classifier, they defined the *generalization error* as the difference between training set and testing set as

   $$\text{GE}(g) = |l_{\text{exp}}(g) - l_{\text{emp}}(g)| \tag{53}$$

   where l_{exp} and l_{emp} is the loss on the training set and testing set respectively. Furthermore, they defined the terms **score** ($o(s_i)$) and **margin** ($\gamma^d(s_i)$) as the following.

$$o(s_i) = \min_{j \neq y_i} \sqrt{2}(\boldsymbol{\delta}_{y_i} - \boldsymbol{\delta}_j)^\top f(\boldsymbol{x}_i) \tag{54}$$

$$\gamma^d(s_i) = \sup\{a : d(\boldsymbol{x}_i, \boldsymbol{x}) \leq a \Rightarrow g(\boldsymbol{x}) = y_i \ \forall \boldsymbol{x}\} \tag{55}$$

where s_i is the pair of training sample and its label $(\boldsymbol{x}_i, y_i)$, $\boldsymbol{\delta}_i$ is the Kronecker delta vector with $(\boldsymbol{\delta}_i)_i = 1$. The authors of this paper claimed the following lower bound for the margin as the following

$$\gamma^d(s_i) \geq \frac{o(s_i)}{\sup_{\boldsymbol{x}:\|\boldsymbol{x}-\boldsymbol{x}_i\|_2 \leq \gamma^d(s_i)} \|\boldsymbol{J}(\boldsymbol{x})\|_2} \tag{56}$$

where $\boldsymbol{J}(\boldsymbol{x})$ is the Jacobian of the probabilities with respect to the input; therefore, its dimension is $D \times C$ where D is the dimension of the input and C is the number of classes.

As the lower bound (56) suggests, the margin can be confirmed to be large if the tractable term $\|\boldsymbol{J}(\boldsymbol{x})\|_2$ can be minimize. Hence, this inspired the idea of regularizing the Jacobian. However, calculating the spectral norm of a matrix can be expensive. To circumvent this problem, the authors of this paper suggests penalizing the Frobenious norm instead based on the following relationship between spectral norm and Frobenious norm

$$\frac{1}{\text{rank}(\boldsymbol{J}(\boldsymbol{x}_i))} \|\boldsymbol{J}(\boldsymbol{x}_i)\|_F^2 \leq \|\boldsymbol{J}(\boldsymbol{x}_i)\|_2^2 \leq \|\boldsymbol{J}(\boldsymbol{x}_i)\|_F^2 \tag{57}$$

2. **Frobenius Regularizer (FrobReg)** Penalizes the squared Frobenius norm of the Jacobian of the logits with respect to the input. The only difference from JacReg is that it penalizes the logits instead of probabilities.

$$\mathcal{L}_{FrobReg}(\boldsymbol{x}, y; \theta) = \mathcal{L}(\boldsymbol{x}, y; \theta) + \lambda \|J_g\|_F^2 \tag{58}$$

5.2 Noise Injection

6 Network Architecture and Dataset

6.1 Network Architecture

6.2 Dataset

7 Experimental Methods and Results

8 Conclusion