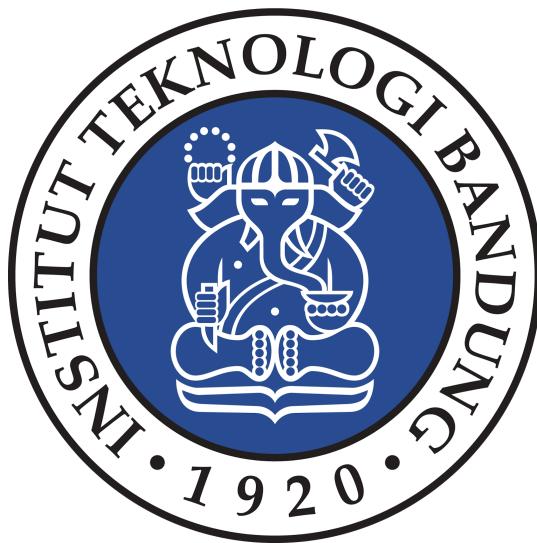


Tugas Besar IF3270 Pembelajaran Mesin
Feedforward Neural Network



Disusun oleh:
Kelompok 18 - K2

Raden Rafly Hanggaraksa B / 13522014

Dhafin Fawwaz Ikramullah / 13522084

Sa'ad Abdul Hakim / 13522092

Program Studi Teknik Informatika

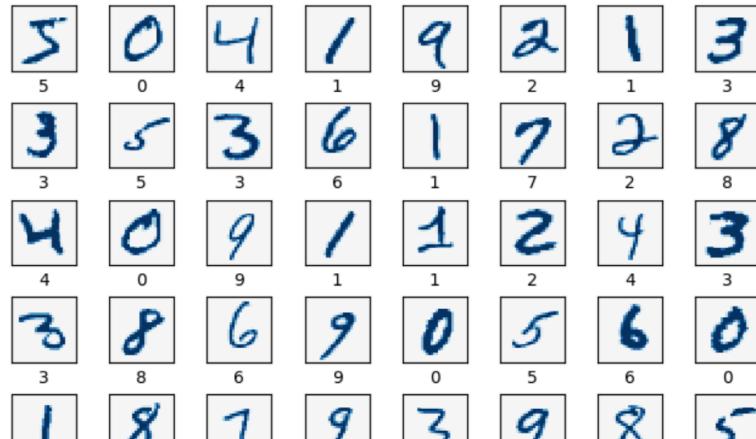
Sekolah Teknik Elektro dan Informatika - Institut Teknologi Bandung

Jl. Ganesha 10, Bandung 40132

2025

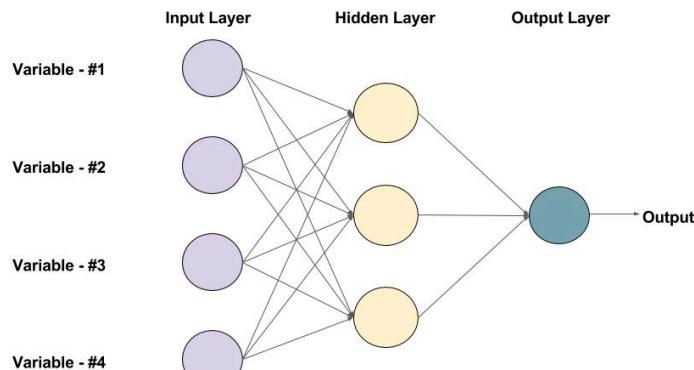
A. Deskripsi Persoalan

MNIST (Modified National Institute of Standards and Technology) adalah dataset yang berisi kumpulan gambar angka tulisan tangan yang digunakan untuk menguji algoritma pengenalan pola dan klasifikasi. Dataset ini terdiri dari gambar berukuran 28×28 piksel dalam skala abu-abu, di mana setiap gambar direpresentasikan sebagai 784 fitur. MNIST dibuat dari gabungan dua dataset NIST, yaitu SD-3 (lebih rapi) dan SD-1 (lebih beragam dan sulit dikenali), dengan total 60.000 gambar untuk pelatihan dan 10.000 gambar untuk pengujian.



Sumber: Dari penulis

Dalam tugas besar ini, dataset MNIST akan digunakan untuk menguji model Feedforward Neural Network (FFNN) yang sudah diimplementasikan dari scratch. Pengujian juga dilakukan untuk melihat pengaruh dari berbagai parameter seperti depth dan width, fungsi aktivasi, learning rate, serta metode inisialisasi bobot. Selanjutnya, hasil prediksi pada pengujian model FFNN yang dibuat juga akan dibandingkan dengan hasil prediksi dengan menggunakan library sklearn MLP.



An example of a Feed-forward Neural Network with one hidden layer (with 3 neurons)

Sumber: <https://learnopencv.com/understanding-feedforward-neural-networks/>

B. Pembahasan

1. Deskripsi Kelas

Class: FFNNClassifier	
Attribute	
hidden_layer_sizes	Banyaknya node setiap hidden layer
X	Data masukan
y	Label target
weights_history	bobot dalam setiap epoch
biases_history	bias dalam setiap epoch
weight_gradients_history	gradien bobot dalam setiap epoch
learning_rate	Laju pembelajaran
activation_func	Fungsi aktivasi untuk setiap layer
max_epoch	Jumlah maksimal iterasi pelatihan
batch_size	Ukuran batch untuk pemrosesan data
loss_func	Fungsi untuk menghitung kesalahan prediksi
init_method	Metode inisialisasi bobot
lower_bound, upper_bound	Batas bawah dan atas untuk inisialisasi bobot
mean	Rata-rata untuk inisialisasi normal
std	Standar deviasi untuk inisialisasi normal
seed	Seed untuk reproduktibilitas
amount_of_features	Jumlah fitur
loss_history	Riwayat training loss dalam setiap epoch
validation_loss_history	Riwayat validation loss dalam setiap epoch
Fungsi	

<code>_generate_initiator_weights()</code>	Menginisialisasi bobot awal dengan metode yang dipilih
<code>_activation_function(x, func)</code>	Implementasi fungsi aktivasi
<code>_activation_derived_function(x, func)</code>	Implementasi setiap turunan fungsi aktivasi
<code>_loss_function(y_act, y_pred, func)</code>	Implementasi fungsi loss yang ada
<code>_get_amount_of_features()</code>	Mengembalikan jumlah fitur
<code>_get_number_of_classes()</code>	Mengembalikan jumlah kelas
<code>_get_hidden_layer_sizes()</code>	Mengembalikan banyaknya node setiap layer
<code>_generate_new_empty_layers()</code>	Membuat struktur layer kosong untuk jaringan
<code>_get_number_of_classes()</code>	Mengembalikan jumlah kelas
<code>fit(X, y)</code>	Melakukan pelatihan model FFNN
<code>preprocess_x(X)</code>	Normalisasi data masukan
<code>preprocess_y(y)</code>	Pengkodean one-hot untuk label
<code>preprocess(X, y)</code>	Menerapkan preproses pada X dan y
<code>predict(X)</code>	Memprediksi kelas untuk data baru
<code>predict_proba(X)</code>	Menghasilkan probabilitas prediksi untuk setiap kelas
<code>save(filename)</code>	Menyimpan model terlatih ke file
<code>load(path)</code>	Memuat model yang tersimpan dari file

Class: WeightInitialization	
Attribute	
<code>layer_units</code>	List jumlah unit di setiap layer
<code>init_method</code>	Metode inisialisasi bobot
<code>lower_bound, upper_bound</code>	Batas bawah dan atas untuk inisialisasi bobot
<code>mean</code>	Rata-rata untuk inisialisasi normal

std	Standar deviasi untuk inisialisasi normal
seed	Seed untuk reproduktibilitas
dtype	Tipe data
coefs_	List vektor intersep (bias)
intercepts_	List vektor intersep (bias)
Fungsi	
_init_coef(fan_in, fan_out)	Menginisialisasi matriks koefisien (bobot) dan vektor intersep (bias) untuk satu layer
initialize_weights()	Menghasilkan bobot dan bias untuk seluruh layer FFNN

Class: NeuralNetworkVisualizerPlotly	
Attribute	
layers	Jumlah node di setiap layer FFNN
weights	Bobot koneksi antar node
gradients	Gradien dari bobot-bobot yang ada
biases	Nilai bias untuk setiap layer
loss_history	Riwayat training loss setiap epoch
validation_loss_history	Riwayat validation loss setiap epoch
colors	Warna untuk setiap layer
layer_names	Nama-nama setiap layer
Fungsi	
_generate_layer_names	Membangkitkan lama tiap layer untuk label pada visualisasi.
plot_network	Memvisualisasikan struktur FFNN dalam bentuk graf
plot_weight_distribution(layers_to_plot)	Membuat histogram distribusi bobot

plot_gradient_distribution(layers_to_plot)	Membuat histogram distribusi gradien
plot_loss	Membuat grafik loss dalam bentuk line dari setiap epoch

2. Penjelasan implementasi

2.1. Inisialisasi Training

Semua variabel yang diperlukan akan diinisialisasi terlebih dahulu dengan numpy dan list python jika size/shapenya berbeda-beda. Dilakukan juga error checking agar yang menggunakan fungsi ini tidak ceroboh dengan parameter yang seharusnya tidak bisa digunakan. Dilakukan pula inisialisasi weight.

```
def fit(self, X: NDArray, y: NDArray):
    if type(y) == list and type(y[0]) != list:
        y = np.array([[i] for i in y], dtype="int32")
    if type(y) == list and type(y[0]) == list:
        y = np.array(y, dtype="int32")

    if len(X) != len(y):
        raise Exception("length of X and y is not the same")
    if len(X) == 0:
        raise Exception("len(self.X) == 0")
    if len(X[0]) == 0:
        raise Exception("len(self.X[0]) == 0")
    if len(y) == 0:
        raise Exception("len(self.y) == 0")

    self.amount_of_features = len(X[0])
    self.X: NDArray = []
    self.y: ArrayLike = []
    self.weights_history: list[NDArray] = []
    self.biases_history: list[ArrayLike] = []
    self.weight_gradients_history: list[NDArray] = []
    self.loss_history = []

    self.X = X.astype("float32")
    self.y = y.astype("int32")
    initial_weight, initial_bias = self._generate_initiator_weights()
```

```
    initial_gradients = [np.zeros_like(w, dtype="float32") for w in
initial_weight]
    self.weights_history = initial_weight
    self.biases_history = initial_bias
    self.weight_gradients_history = initial_gradients
    layer_sizes = self._get_hidden_layer_sizes()
    network_depth = len(layer_sizes)
```

2.2. Forward Propagation

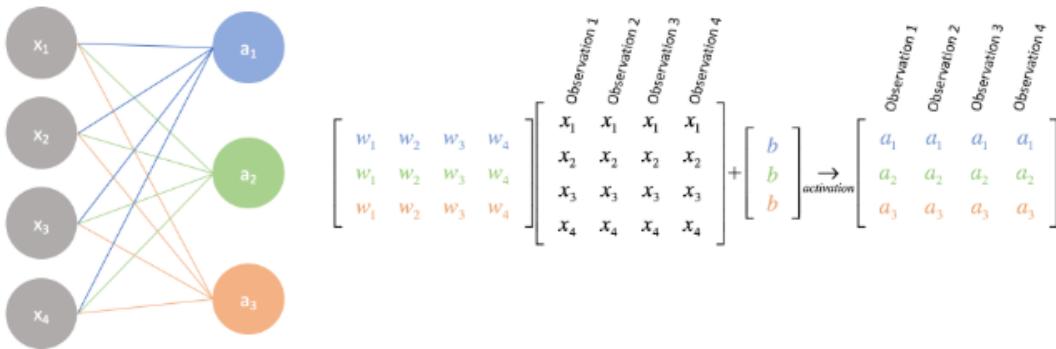
Sebelum memulai forward propagation dilakukan dulu iterasi untuk setiap epoch dan untuk setiap batch size input. Berikut ini penjelasan untuk setiap variabel penting ini.

- weight: list of weight numpy matrix yang menghubungkan 2 layer
- nodes: list of value numpy matrix
- nodes_active: list of value numpy matrix hasil fungsi aktivasi
- biases: list of bias numpy array

Tiap elemen ke i dari list menyatakan layer ke i. Di dalamnya terdapat matrix yang menyatakan semua node atau edge pada sebuah layer.

```
for epoch in range(self.epoch_amount):
    total_loss = 0
    current_dataset_idx = 0
    while current_dataset_idx < len(self.X):
        weights, nodes, nodes_active, biases =
self._generate_new_empty_layers()
```

until_idx di sini hanya untuk memastikan jika batch_size tidak habis membagi jumlah instance data training. Untuk layer pertama tidak ada kalkulasi apapun, hanya mengambil dari data training saja termasuk untuk nodes_active. Lalu untuk setiap layer, akan dihitung perkalian matrix antara value matrix di layer sebelumnya dengan weight matrix di layer sebelumnya ditambah bias matrix di layer sebelumnya. Bisa dilihat bahwa di sini dimensi dari bias matrix tidak akan bersesuaian dengan hasil perkalian matrix weight dan value. Tetapi di sini numpy akan secara otomatis membroadcast value dari bias matrix sehingga value di bias matrix akan diduplikan untuk menyamakan dimensi dari matrix hasil perkalian weight matrix dan value matrix. Setelah itu hasilnya disimpan di nodes karena akan digunakan saat back propagation. Hasilnya juga akan dimasukkan ke dalam fungsi aktivasi dan disimpan untuk epoch berikutnya. Berikut ini visualisasi untuk perkalian matrixnya untuk perkalian yang terjadi antar 2 layer.



Sumber: <https://www.jeremyjordan.me/intro-to-neural-networks/>

Perkalian pada visualisasi di atas akan dilakukan pada setiap layer dari hidden layer pertama sampai ke output layer.

```

until_idx = min(current_dataset_idx+batch_size, len(self.X))
nodes[0] = self.X[current_dataset_idx:until_idx]
nodes_active[0] = self.X[current_dataset_idx:until_idx] # not passed to
activation function for the first layer

for k in range(1, network_depth):
    w_k = self.weights_history[k-1]
    b_k = self.biases_history[k-1]
    h_k_min_1 = nodes_active[k-1]
    a_k = b_k + (h_k_min_1 @ w_k) # numpy will automatically broadcast
b_k (row will be copied to match the result from dot) so that this is addable

    nodes[k] = a_k
    nodes_active[k] = FFNNClassifier._activation_function(a_k,
self.activation_func[k-1])

```

2.3. Loss Calculation

Sesuai dengan spesifikasi, loss akan dihitung dan disimpan untuk diprint jika ingin menjalankan training secara verbose dan juga untuk keperluan visualisasi. y_{act} di sini adalah nilai target, y_{pred} di sini adalah hasil prediksi forward propagation yang baru saja dilakukan. Nilainya sama dengan value pada node di layer terakhir (output layer). Lalu dihitung lossnya. Loss ini kemudian akan dijumlahkan ke total loss. Nanti di akhir sebelum berpindah ke epoch selanjutnya total loss akan dibagi dengan jumlah instance data training.

```

y_act = self.y[current_dataset_idx:until_idx]
y_pred = nodes_active[network_depth-1]

```

```

        loss = FFNNClassifier._loss_function(
            y_act=y_act,
            y_pred=y_pred,
            func=self.loss_func
        )
        total_loss += loss * (until_idx - current_dataset_idx)
    
```

2.4. Backward Propagation

Backward propagation diawali dengan inisialisasi terlebih dahulu variabel yang diperlukan yaitu list of gradien weight dan gradien bias. Untuk kasus khusus yaitu jika:

- Activation function softmax dan loss function categorical cross entropy, atau
- Activation function sigmoid dan loss function binary cross entropy, atau
- Activation function linear dan loss function mean squared error

Maka hasil turunan untuk mendapatkan deltananya akan sesederhana mengurangkan matrix nilai dengan target. Jika activation functionnya softmax dan loss functionnya bukan categorical cross entropy, maka akan dilakukan perlakuan khusus. Sangat tidak disarankan untuk menggunakan kombinasi ini. Hasilnya tidak akan baik. Selain kasus sebelumnya, maka akan digunakan fungsi yang sudah dibuat. Tetapi ini juga tidak disarankan. Gunakanlah kombinasi pada branch if pertama.

```

    weight_gradiens = [0 for i in range(len(self.weights_history))] # 0
will be replaced with numpy.array
    bias_gradiens = [0 for i in range(len(self.biases_history))] # 0 will
be replaced with numpy.array

    delta = 0
    if (self.activation_func[-1] == 'softmax' and self.loss_func ==
'categorical_cross_entropy') or (self.activation_func[-1] == "sigmoid" and
self.loss_func == 'binary_cross_entropy') or (self.activation_func[-1] ==
"linear" and self.loss_func == 'mean_squared_error'):
        delta = (nodes_active[-1] -
self.y[current_dataset_idx:until_idx]).astype("float32")

    elif self.activation_func[-1] == 'softmax' and self.loss_func !=
'categorical_cross_entropy':
        jacobians = FFNNClassifier._activation_derived_function(nodes[-1],
self.activation_func[-1])

```

```

        loss_grad = FFNNClassifier._loss_function_derived(
            y_act=y_act,
            y_pred=y_pred,
            func=self.loss_func
        )
        loss_grad_col = loss_grad[..., np.newaxis] # (batch_size, n) ->
(batch_size, n, 1)
        delta = np.matmul(jacobians, loss_grad_col) # (batch_size, n, 1)
        delta = np.squeeze(delta, axis=-1) # (batch_size, n)
    else:
        loss_grad = FFNNClassifier._loss_function_derived(
            y_act=y_act,
            y_pred=y_pred,
            func=self.loss_func
        )
        delta = loss_grad *
FFNNClassifier._activation_derived_function(nodes[-1],
self.activation_func[-1])

```

Setelah delta didapatkan, kita bisa mulai melakukan backward propagation. Untuk output layer langsung dihitung saja yaitu matrix nilai di layer terakhir dikali delta dibagi batch size. Untuk bias yaitu rata-rata dari delta. Kemudian lakukan iterasi mulai dari hidden layer terakhir sampai hidden layer pertama. Update delta menjadi perkalian antara delta dan weight matrix yang sudah ditranspose dikali dengan hasil dari turunan fungsi aktivasi dari matrix nilai sebelum dimasukkan ke fungsi aktivasi. Lalu gradien bobot pada hidden layer tersebut adalah matrix nilai di layer tersebut ditranspose kemudian dikali delta dan dibagi batch size. Untuk bias adalah rata-rata dari delta. Lalu simpan weight gradiennya sesuai keperluan spek untuk divisualisasikan.

```

        weight_gradiens[network_depth-2] = nodes_active[k-1].T @ delta /
self.batch_size
        bias_gradiens[network_depth-2] = np.mean(delta, axis=0, keepdims=True)

        for k in range(network_depth-2, 0, -1): # from the last hidden layer
(not including the output layer)
            w = self.weights_history[k]

            delta = np.dot(delta, w.T) *
FFNNClassifier._activation_derived_function(nodes[k],

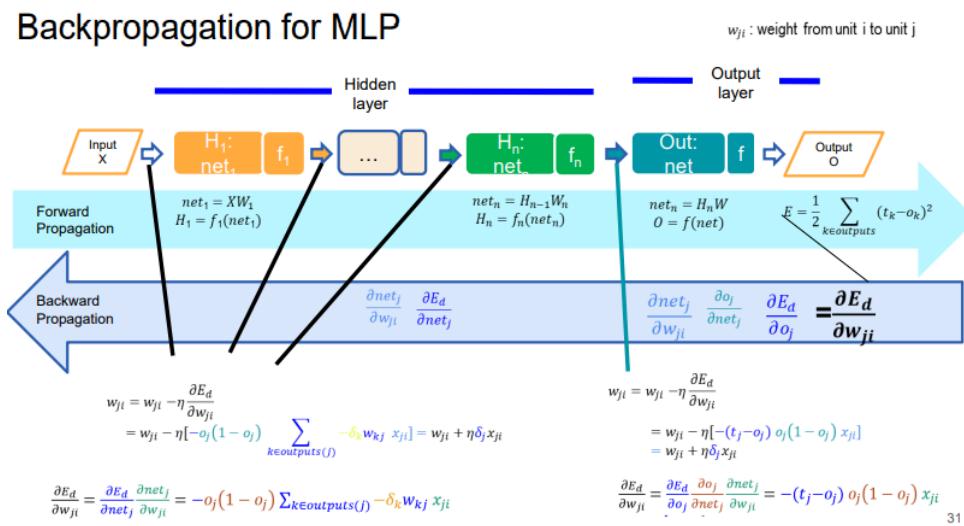
```

```

        self.activation_func[k-1])
            weight_gradiens[k-1] = nodes_active[k-1].T @ delta /
self.batch_size
            bias_gradiens[k-1] = np.mean(delta, axis=0, keepdims=True)

        self.weight_gradients_history = weight_gradiens
    
```

Proses back propagation ini dilakukan berdasarkan konsep turunan. Secara visualisasi bisa dilihat gambar berikut.



Sumber:

https://edunexcontentprodhot.blob.core.windows.net/edunex/nullfile/1741577891298_IF3270-Mgg03-FFNN-print?sv=2024-11-04&spr=https&st=2025-03-10T03%3A32%3A33Z&se=2027-03-10T03%3A32%3A33Z&sr=b&sp=r&sig=KdLqCzzo5Z3lQwFtxt%2BEoNGBpLkPPB2YMdYxbAAqYzQ%3D&rsct=application%2Fpdf

2.5. Weight Update

Setelah backpropagation selesai, dilakukan updating bobot. Proses ini cukup sederhana yaitu untuk setiap layer, update bobotnya menjadi bobot sebelumnya dikurang dengan perkalian antara learning rate dan gradien bobot di layer tersebut. Lakukan hal yang sama untuk bias. Setelah itu simpan bobot dan biasnya dan maju ke iterasi batch berikutnya. Setelah semua batch data training sudah diproses, seperti yang sudah dijelaskan sebelumnya, total loss akan dibagi dengan jumlah instance data training lalu disimpan. Setelah itu kita bisa mulai lanjut ke epoch berikutnya dan ulangi step sebelumnya.

```

for k in range(network_depth-1):
    
```

```

        w_k = self.weights_history[k]
        b_k = self.biases_history[k]

        weights[k] = w_k - self.learning_rate * weight_gradiens[k]
        biases[k] = b_k - self.learning_rate * bias_gradiens[k]

        self.weights_history = weights
        self.biases_history = biases

        current_dataset_idx += self.batch_size

        current_loss = total_loss / len(self.X)
        self.loss_history.append(current_loss)
        if self.verbose == 1:
            print(f"Epoch {epoch+1}/{self.epoch_amount} done, loss: {current_loss}")

        elif self.verbose == 2:
            print("====")
            print(f"Epoch {epoch+1}/{self.epoch_amount} done")
            print(f"weights: {self.weights_history}")
            print(f"biases: {self.biases_history}")

    return self.loss_history

```

2.6. Inferensi

Untuk melakukan inferensi sebenarnya sama saja dengan forward propagation. Lakukan terlebih dahulu forward propagation untuk mendapatkan list nilai di output layer. List nilai ini akan berisi 10 elemen yang berisikan probabilitas setiap kemungkinan output. Pilih elemen yang memiliki output terbesar. Index dari elemen tersebut lah yang merupakan hasil prediksinya.

```

def predict(self, X_test: NDArray):
    proba = self.predict_proba(X_test)
    return np.argmax(proba, axis=1)

def predict_proba(self, X_test: NDArray):
    prediction = np.zeros((len(X_test), self._get_number_of_classes()))
    current_idx = 0
    while current_idx < len(X_test):

```

```

        weights, nodes, nodes_active, biases =
self._generate_new_empty_layers()
        until_idx = min(current_idx+batch_size, len(X_test))
nodes[0] = X_test[current_idx:until_idx]
nodes_active[0] = X_test[current_idx:until_idx]

        for k in range(1, len(self.weights_history)+1):
            w_k = self.weights_history[k-1]
            b_k = self.biases_history[k-1]
            h_k_min_1 = nodes_active[k-1]

            a_k = b_k + np.dot(h_k_min_1, w_k)

            nodes[k] = a_k
            nodes_active[k] = FFNNClassifier._activation_function(a_k,
self.activation_func[k-1])

            prediction[current_idx:until_idx] = nodes_active[-1]
            current_idx += self.batch_size
return prediction

```

2.7. Bonus: 2 Fungsi Aktivasi

Fungsi aktivasi yang diimplementasikan untuk bonus yaitu softsign dan softplus berdasarkan <https://arxiv.org/pdf/1811.03378v1.pdf>. Berikut ini implementasinya.

Fungsi	Implementasi	Turunan Pertama
softsign	$f(x) = x / (1 + x)$	$f'(x) = 1/(1+ x)^2$
softplus	$f(x) = \ln(1 + e^x)$	$f'(x) = 1 / (1 + e^{-x})$

3. Hasil pengujian

3.1 Setup Pengujian

Untuk pengujian akan dibandingkan nilai dari weight, bias, prediksi, probabilitas prediksi, loss, dan akurasi. MLPClassifier pada sklearn sendiri tidak menyediakan cara untuk menginisialisasi initial bobot. Maka dibuat class untuk meng override inisialisasi bobot untuk weight dan bias. Yang juga dilakukan adalah untuk inisialisasi seed agar hasil randomnya tetap sama. Jadi fungsi MLPClassifier yang dioverride adalah `_initialize()` dan `_init_coef()`. `_initialize()` sebenarnya hanya untuk menyelipkan code untuk set seed. `_init_coef()` untuk memunculkan opsi untuk menggunakan inisialisasi zero, uniform, dan normal. Setelah dilakukan pengujian, FFNN yang dibuat sendiri memiliki hasil yang sama seperti MLPClassifier dari SKLearn terutama untuk bobot akhir, bias akhir, prediksi, probabilitas prediksi, loss, dan akurasi. Berikut ini outputnya.

```
[SKLearn MLPClassifier]
Iteration 1, loss = 1.16667774
Iteration 2, loss = 1.15985784
Iteration 3, loss = 1.15372844
D:\ITB\Semester 6\ML\Tubes
1\venv\Lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:692: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (3) reached and the optimization hasn't converged yet.
  warnings.warn(
Weights:
[array([[ -0.70295755,   0.17493439,   0.53065978,   0.46476807],
       [ 0.37534561,   0.30354627,  -0.09887265,  -0.22962205],
       [-0.36951244,   0.51963575,  -0.58463959,  -0.0546431 ]]),
array([[ 0.11751219,   0.41163474,  -0.515175 ],
       [ 0.32618261,  -0.48908488,  -0.87232367],
       [-0.52469752,   0.07443457,  -0.92359425],
       [ 0.00883137,   0.13310468,  -0.42268854]]), array([[-0.2861781,
       , -0.51613803],
       [-1.00407131,  -0.29502086],
       [-0.12041679,   0.60433127]]), array([[[-0.106268 ,
       -0.04289961,   0.05939749],
       [ 0.31938053,  -0.69037129,  -0.6041134 ]]])
Biases:
[array([ 0.4030178 , -0.26532339,   0.51143754,  -0.26448829]),
array([-0.21756329,  -0.85628103,  -0.81394763]), array([0.7017874 ,
0.14435068]), array([-0.12853568,  -0.06363935,  -0.56830764])]
Prediction:
[0 0 0 0 0]
Prediction Probability:
[[0.46615546 0.32047734 0.2133672 ]
[0.46688677 0.3199319  0.21318133]
[0.4662833  0.3203425  0.21337419]]
```

```

[0.46631111 0.32032863 0.21336027]
[0.46656518 0.32010797 0.21332685]
[0.46650813 0.32015756 0.21333432]]

Loss:
[np.float64(1.1666777354829996), np.float64(1.1598578412590186),
np.float64(1.1537284396796583)]

Accuracy:
0.3333333333333333

[From Scratch FFNNClassifier]
Epoch 1/3 done, loss: 1.1666777115087266
Epoch 2/3 done, loss: 1.1598578345881683
Epoch 3/3 done, loss: 1.1537284749443038

Weights:
[array([[ -0.7029575 ,  0.17493439,  0.53065974,  0.46476808],
       [ 0.37534562,  0.30354628, -0.09887265, -0.22962205],
       [-0.36951244,  0.51963574, -0.5846396 , -0.0546431 ]],
      dtype=float32), array([[ 0.1175122 ,  0.4116347 , -0.515175 ],
       [ 0.3261826 , -0.4890849 , -0.8723237 ],
       [-0.52469754,  0.07443457, -0.9235942 ],
       [ 0.00883137,  0.13310467, -0.42268854]], dtype=float32),
array([[[-0.28617808, -0.516138 ],
       [-1.0040714 , -0.29502085],
       [-0.12041679,  0.60433125]], dtype=float32), array([[-0.106268,
       -0.04289961,  0.05939749],
       [ 0.31938055, -0.6903713 , -0.60411346]], dtype=float32)]
Biases:
[array([[ 0.40301782, -0.2653234 ,  0.5114376 , -0.26448828]],
      dtype=float32), array([[-0.21756329, -0.85628104, -0.8139476
]], dtype=float32), array([[0.70178735, 0.14435068]], dtype=float32),
array([[-0.12853567, -0.06363935, -0.56830764]], dtype=float32)]

Prediction:
[0 0 0 0 0]

Prediction Probability:
[[0.46615547 0.32047734 0.21336719]
[0.46688678 0.3199319 0.21318132]
[0.46628331 0.3203425 0.21337419]
[0.46631111 0.32032863 0.21336026]
[0.46656519 0.32010797 0.21332684]
[0.46650813 0.32015756 0.21333431]]

Loss:
[np.float64(1.1666777115087266), np.float64(1.1598578345881683),
np.float64(1.1537284749443038)]

Accuracy:
0.3333333333333333

```

```
[Comparison Result]
```

- ✓ Weight is equal
- ✓ Bias is equal
- ✓ Prediction is equal
- ✓ Prediction Probability is equal
- ✓ Loss is equal
- ✓ Accuracy is equal

Yang membedakan pada output ini hanyalah loss. Tetapi sebenarnya sama dan justru lebih akurat. Pada FFNN yang dibuat sendiri menggunakan float64 yang bisa sampai 16 decimal places. Sementara MLPClassifier sklearn menggunakan float32 yang hanya bisa sampai 8 decimal places. Berikut ini contoh dengan data training yang lebih banyak yaitu dengan MNIST.

```
[SKLearn MLPClassifier]
```

```
Iteration 1, loss = 2.01863161
Iteration 2, loss = 1.56106977
Iteration 3, loss = 1.24746778
Iteration 4, loss = 0.92882408
Iteration 5, loss = 0.74729214
Iteration 6, loss = 0.63998837
Iteration 7, loss = 0.56717769
Iteration 8, loss = 0.51661487
Iteration 9, loss = 0.47911499
Iteration 10, loss = 0.44980274
Iteration 11, loss = 0.42602082
Iteration 12, loss = 0.40542195
Iteration 13, loss = 0.38760791
Iteration 14, loss = 0.37210657
Iteration 15, loss = 0.35815323
D:\ITB\Semester 6\ML\Tubes
1\venv\Lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:692: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (15) reached and the optimization hasn't converged yet.
  warnings.warn(
Accuracy:
  0.8920714285714286
```

```
[From Scratch FFNNClassifier]
```

```
Epoch 1/15 done, loss: 2.018631606238401
Epoch 2/15 done, loss: 1.5610697725198983
Epoch 3/15 done, loss: 1.2474677805175758
Epoch 4/15 done, loss: 0.9288240834738939
Epoch 5/15 done, loss: 0.7472921445542219
Epoch 6/15 done, loss: 0.6399883706967543
```

```

Epoch 7/15 done, loss: 0.5671776889333173
Epoch 8/15 done, loss: 0.5166148643850709
Epoch 9/15 done, loss: 0.47911498905552624
Epoch 10/15 done, loss: 0.44980274126295244
Epoch 11/15 done, loss: 0.4260208155096058
Epoch 12/15 done, loss: 0.40542195004254244
Epoch 13/15 done, loss: 0.387607905233625
Epoch 14/15 done, loss: 0.3721065642094826
Epoch 15/15 done, loss: 0.35815323127639326
Accuracy:
0.8920714285714286

```

[Comparison Result]

- Weight is equal
- Bias is equal
- Prediction is equal
- Prediction Probability is equal
- Loss is equal
- Accuracy is equal

Kesimpulan dari setup ini adalah implementasi FFNN sudah bedar dengan bukti bahwa output memiliki nilai yang sama dengan MLPClassifier SKLearn termasuk final weight, final bias, prediksi, probabilitas prediksi, loss, dan akurasi.

3.2 Pengaruh depth dan width

Dari hasil pengujian dengan berbagai konfigurasi depth:

Kedalaman	Library	Implementasi
(10,10)	0.6517	0.6517
(10,10,10)	0.4722	0.4722
(10,10,10,10)	0.2403	0.2403

Berdasarkan hasil tersebut, model dengan dua lapisan tersembunyi ([10, 10]) memperoleh akurasi sekitar 65.17%, yang merupakan hasil terbaik dalam eksperimen ini. Namun, ketika jumlah lapisan tersembunyi ditambah menjadi tiga ([10, 10, 10]), akurasi menurun menjadi 47.22%, dan ketika ditambah lagi menjadi empat lapisan tersembunyi ([10, 10, 10, 10]), akurasi semakin menurun drastis hingga 24.03%.

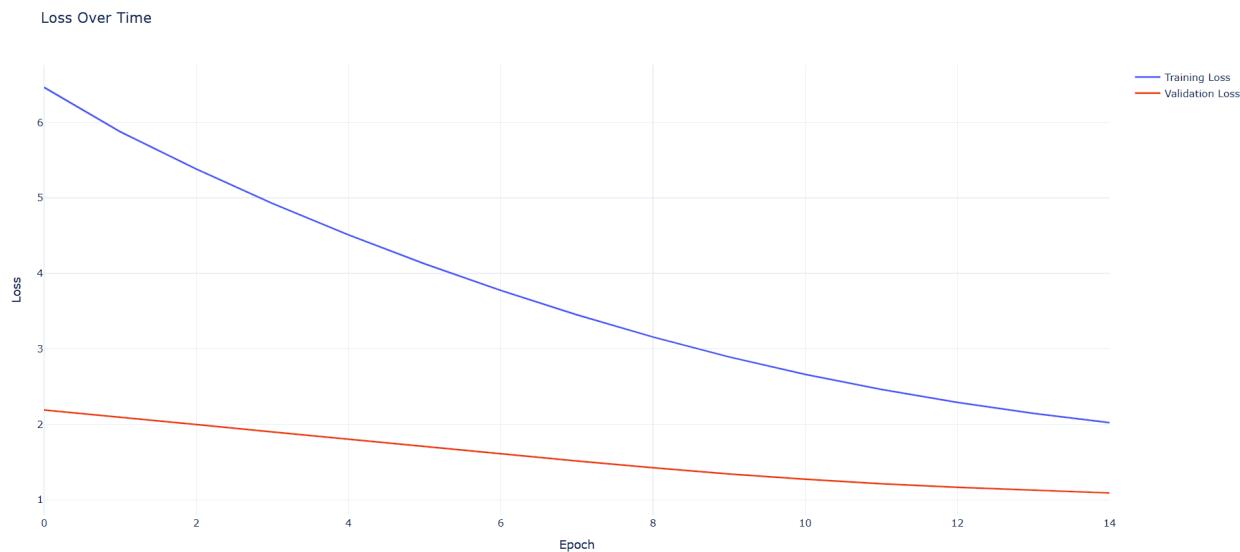
Penurunan akurasi ini kemungkinan besar disebabkan oleh vanishing gradient problem, terutama karena model menggunakan fungsi aktivasi sigmoid, yang cenderung menyebabkan pembaruan bobot menjadi sangat kecil di lapisan terdalam. Akibatnya, semakin dalam jaringan, semakin sulit bagi model untuk melakukan pembelajaran yang efektif. Selain itu, model yang terlalu

dalam juga lebih rentan terhadap overfitting, yang bisa menyebabkan kinerja yang buruk pada data uji.

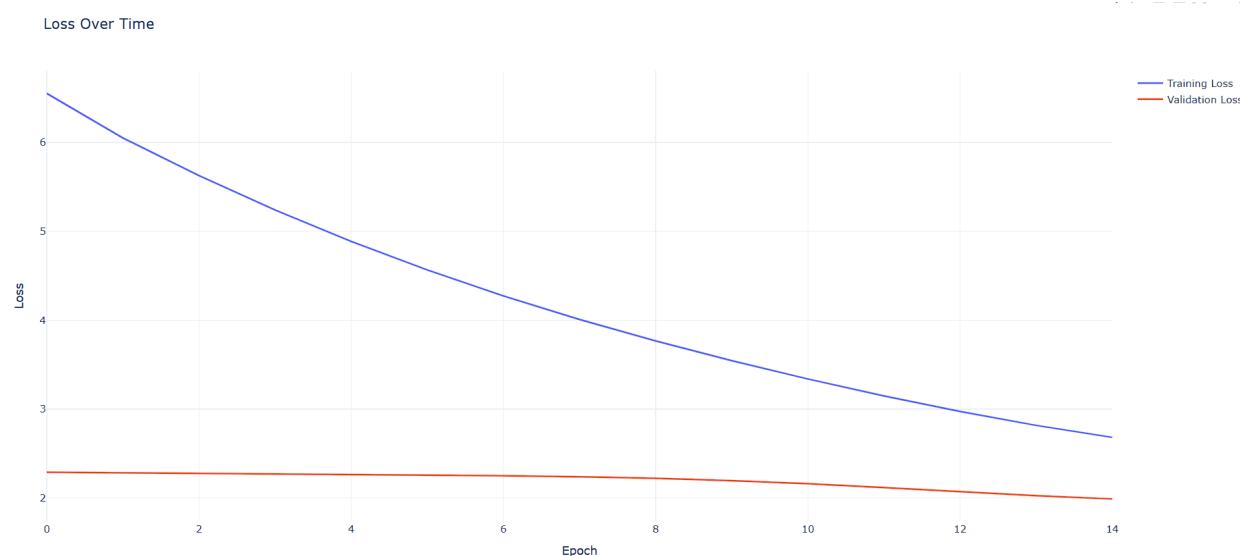
Menariknya, hasil dari implementasi manual dan Sklearn MLPClassifier identik, yang menunjukkan bahwa implementasi FFNN manual sudah benar dan mampu meniru perilaku dari model jaringan saraf buatan bawaan Sklearn. Hal ini juga mengindikasikan bahwa penurunan akurasi memang disebabkan oleh keterbatasan arsitektur jaringan, bukan oleh perbedaan implementasi.

Grafik loss pengujian berbagai konfigurasi depth terurut sesuai dengan tabel diatas:

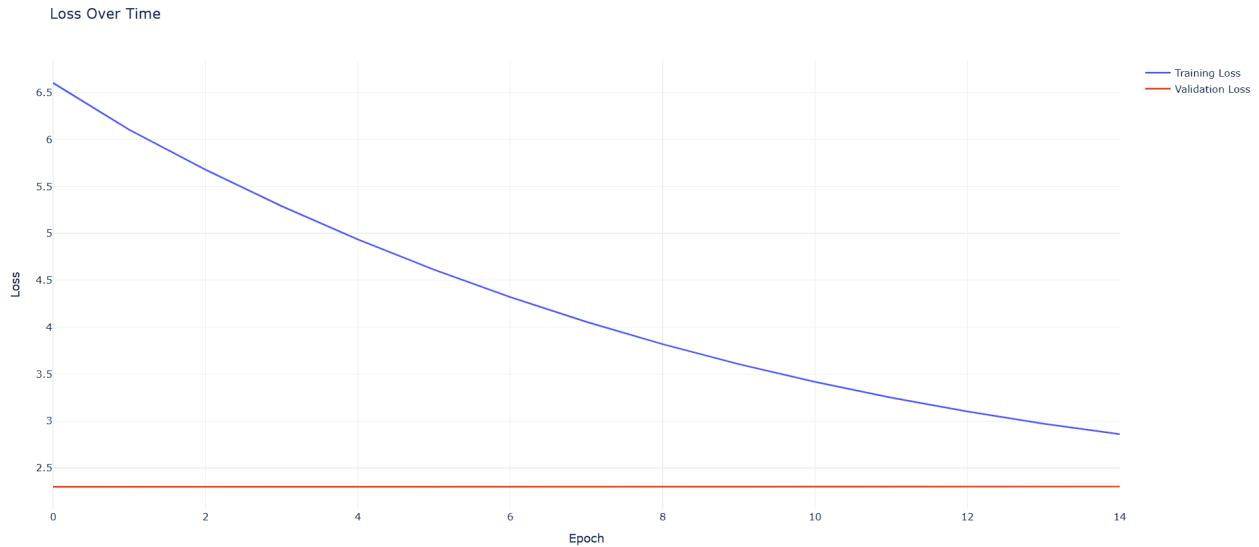
- Kedalaman (10,10)



- Kedalaman (10,10,10)



- Kedalaman (10,10,10,10)



Secara umum, grafik yang dihasilkan memiliki bentuk yang sama terutama pada nilai Training Loss yang cenderung menurun, yang berarti model semakin baik dalam mempelajari data pelatihan. Namun, disisi lain nilai Validation Loss tidak selalu ikut menurun, bahkan cenderung stagnan atau memburuk seiring bertambahnya hidden layer. Hal ini mengindikasikan bahwa peningkatan jumlah hidden layer dapat meningkatkan risiko overfitting.

Dari hasil pengujian dengan berbagai konfigurasi width:

Kelebaran	Library	Implementasi
5	0.7432	0.7432
15	0.8022	0.8022
30	0.8433	0.8433

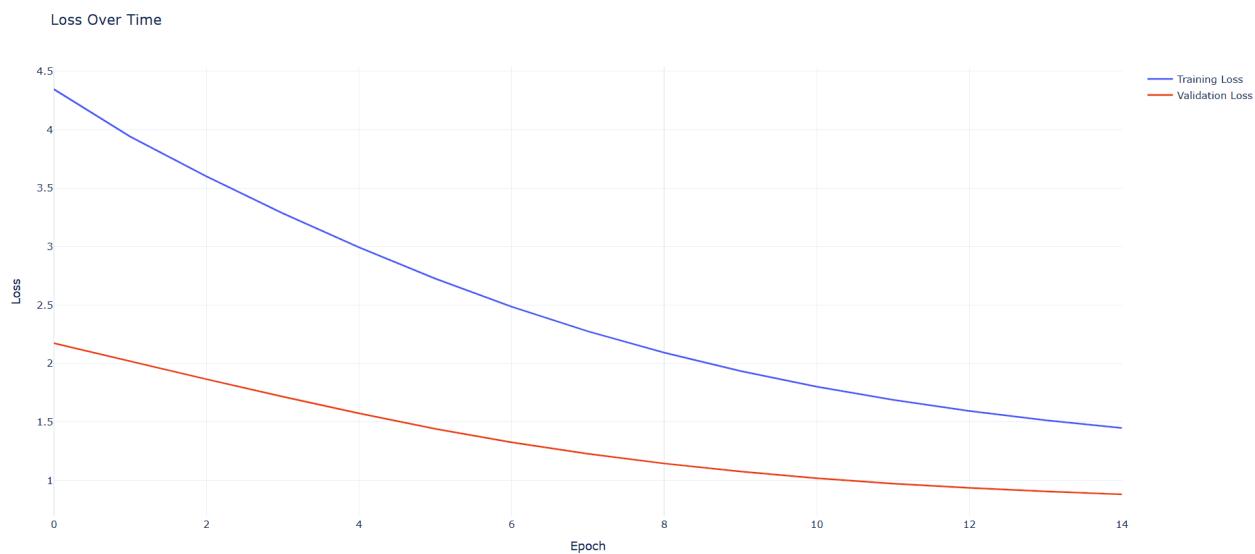
Hasil pengujian menunjukkan bahwa peningkatan jumlah neuron dalam satu hidden layer berdampak positif terhadap akurasi model. Dengan konfigurasi 5 neuron per layer, model memperoleh akurasi sebesar 74.32%, sementara saat jumlah neuron ditingkatkan menjadi 15, akurasi naik menjadi 80.22%. Lebih lanjut, ketika jumlah neuron dalam hidden layer diperbanyak menjadi 30, akurasi meningkat menjadi 84.33%.

Peningkatan akurasi ini disebabkan oleh kapasitas representasi jaringan yang lebih besar seiring dengan bertambahnya jumlah neuron. Dengan lebih banyak neuron, model dapat menangkap lebih banyak pola dalam data sehingga menghasilkan prediksi yang lebih baik. Namun, terdapat batasan dalam menambah jumlah neuron, karena jika jumlahnya terlalu besar, model berisiko mengalami overfitting, model terlalu menyesuaikan diri dengan data pelatihan sehingga kinerjanya pada data uji justru menurun. Selain itu, semakin banyak neuron juga menyebabkan waktu komputasi meningkat, karena jumlah parameter yang harus dioptimasi bertambah secara signifikan.

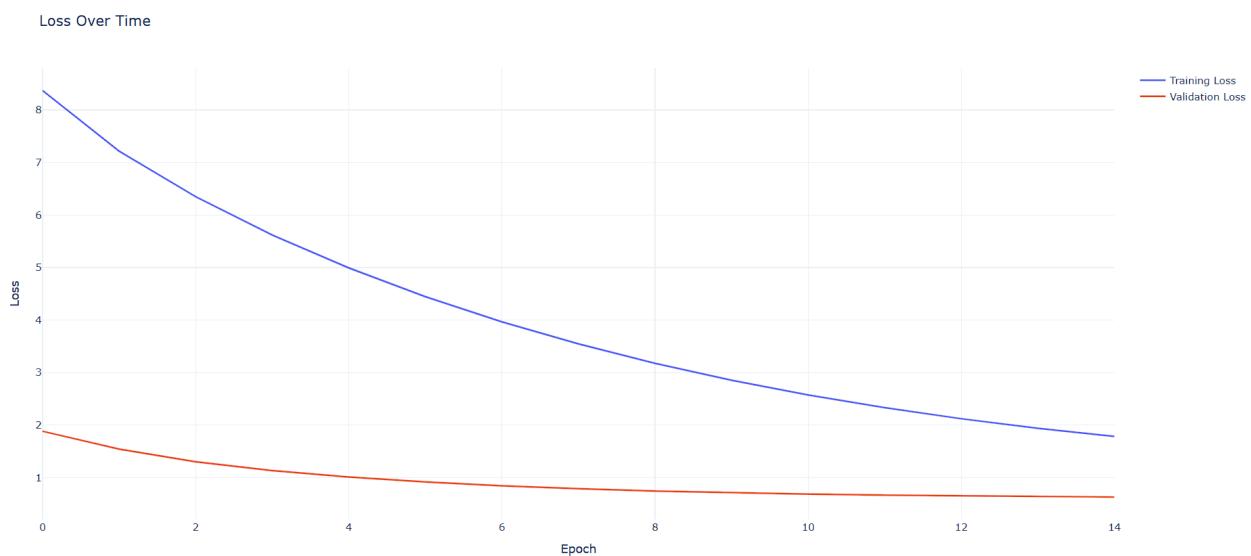
Menariknya, hasil yang diperoleh dari implementasi manual FFNN identik dengan MLPClassifier dari Sklearn, yang menunjukkan bahwa implementasi dari awal telah benar dan mampu mereplikasi perilaku model bawaan Sklearn. Berdasarkan hasil ini, dapat disimpulkan bahwa meningkatkan jumlah neuron dalam hidden layer merupakan strategi yang efektif untuk meningkatkan akurasi model, namun perlu mempertimbangkan risiko overfitting dan efisiensi komputasi.

Grafik loss pengujian berbagai konfigurasi width terurut sesuai dengan tabel diatas:

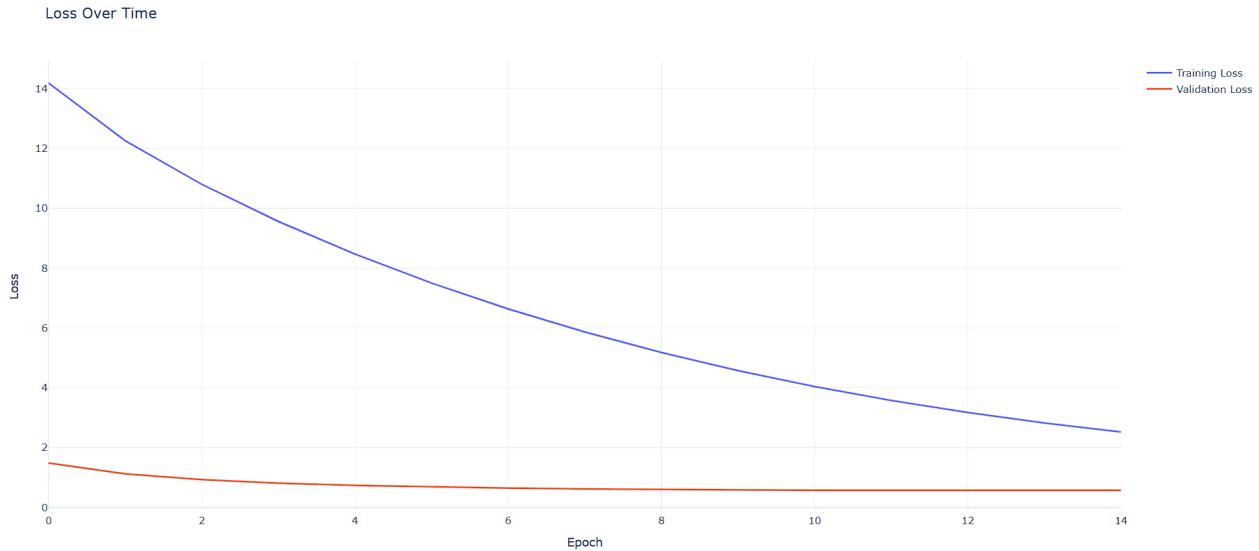
- Kelebaran 5



- Kelebaran 15



- Kelebaran 30



Berdasarkan grafik yang dihasilkan, terlihat bahwa semua model menunjukkan penurunan loss yang signifikan, dengan model yang lebih lebar mencapai Training Loss yang lebih rendah namun menunjukkan gap lebih besar antara Training dan Validation Loss, mengindikasikan risiko overfitting yang meningkat seiring bertambahnya kompleksitas model.

3.3 Pengaruh fungsi aktivasi

Dari hasil pengujian dengan berbagai konfigurasi fungsi aktivasi:

Fungsi Aktivasi	Library	Implementasi
Linear	0.8858	0.8858
ReLU	0.7581	0.7581
Sigmoid	0.8449	0.8449
Tanh	0.8095	0.8095

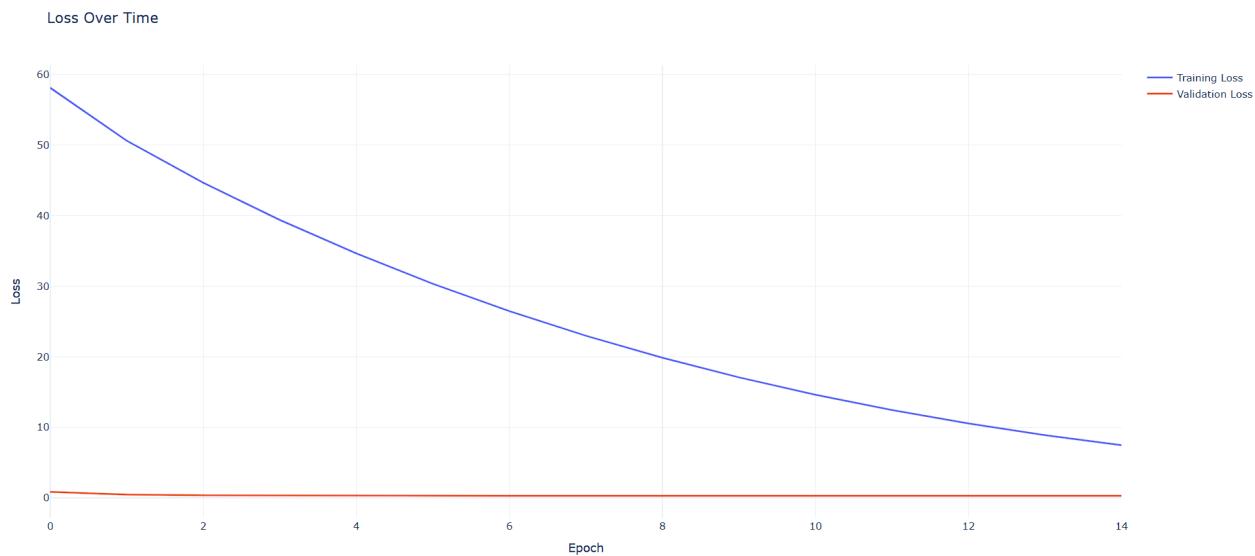
Berdasarkan hasil pengujian dengan berbagai fungsi aktivasi, terlihat bahwa pemilihan aktivasi berpengaruh terhadap performa model. Fungsi aktivasi linear memberikan akurasi tertinggi sebesar 88.58%, yang menunjukkan bahwa dataset ini dapat dipisahkan secara linear dengan cukup baik tanpa memerlukan transformasi non-linear yang kompleks. Di sisi lain, aktivasi ReLU menghasilkan akurasi yang lebih rendah, yaitu 75.81%, kemungkinan karena adanya masalah vanishing gradient pada beberapa unit atau model yang tidak cukup dalam untuk memanfaatkan keuntungan dari ReLU sepenuhnya.

Aktivasi sigmoid mencapai akurasi 84.49%, yang cukup baik tetapi masih dapat mengalami vanishing gradient, terutama saat nilai input jauh dari nol. Aktivasi tanh, yang merupakan versi skala dari sigmoid, memberikan akurasi 80.95%, sedikit lebih baik dari ReLU tetapi masih lebih rendah dibandingkan linear dan sigmoid.

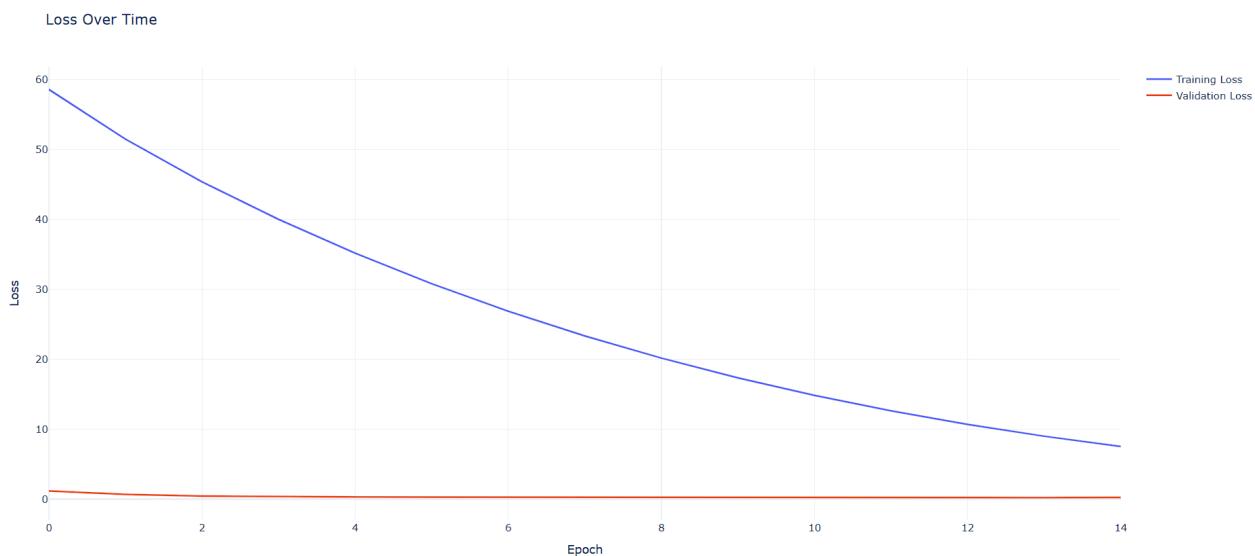
Dari hasil ini, dapat disimpulkan bahwa fungsi aktivasi yang paling cocok untuk dataset ini adalah linear dan sigmoid, karena keduanya memberikan akurasi yang lebih tinggi dibandingkan ReLU dan tanh. Namun, dalam kasus lain dengan data yang lebih kompleks dan tidak dapat dipisahkan secara linear, fungsi aktivasi non-linear seperti ReLU atau tanh mungkin lebih unggul.

Grafik loss pengujian berbagai konfigurasi fungsi aktivasi terurut sesuai dengan tabel diatas:

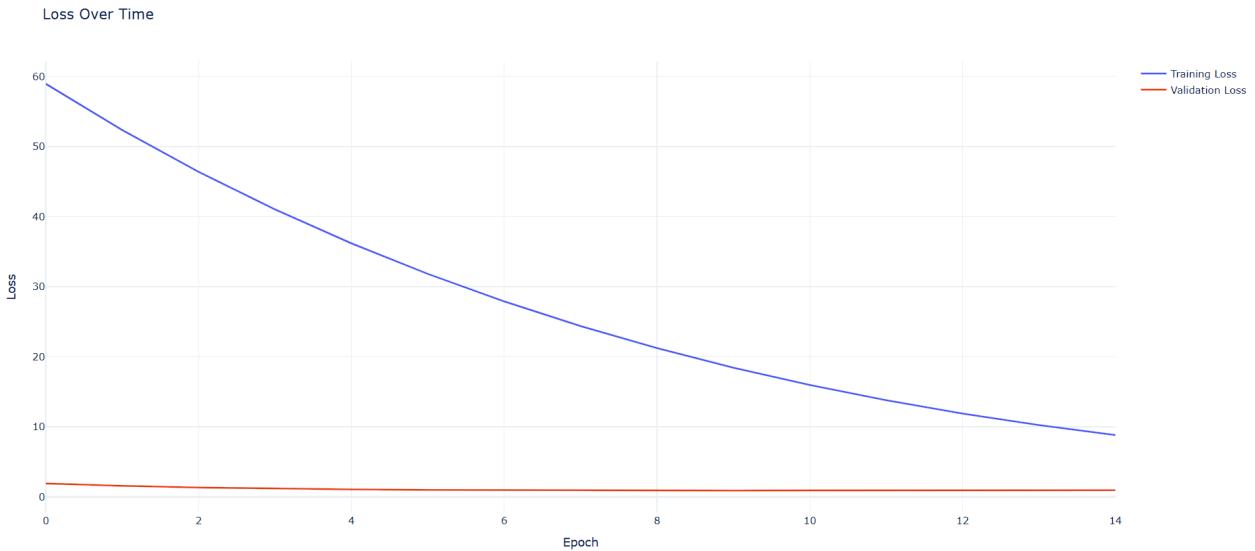
- Linear



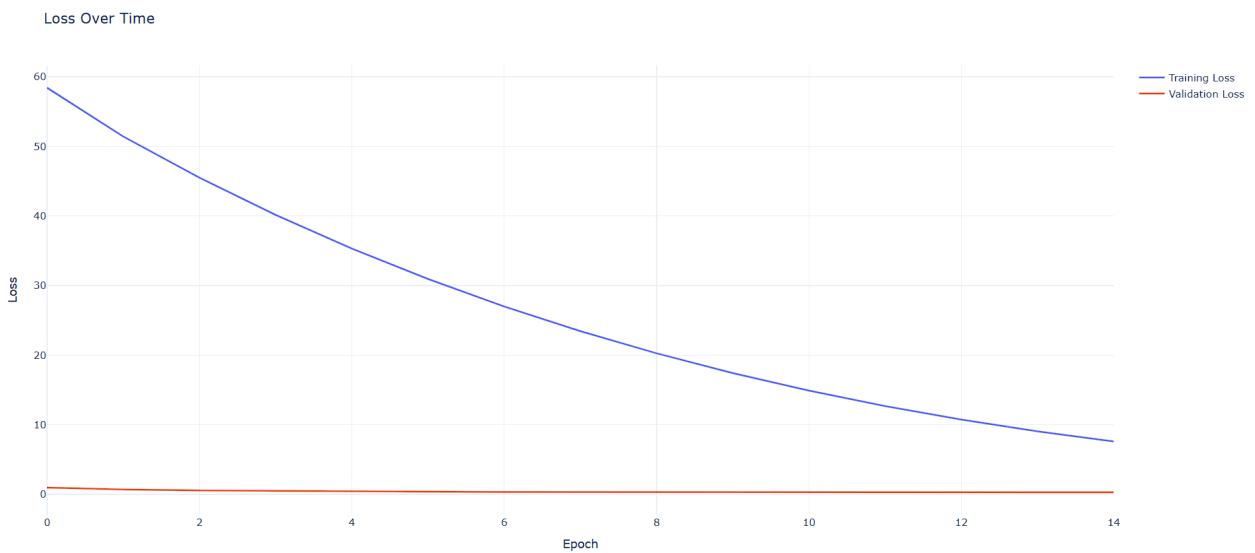
- ReLU



- Sigmoid



- Tanh

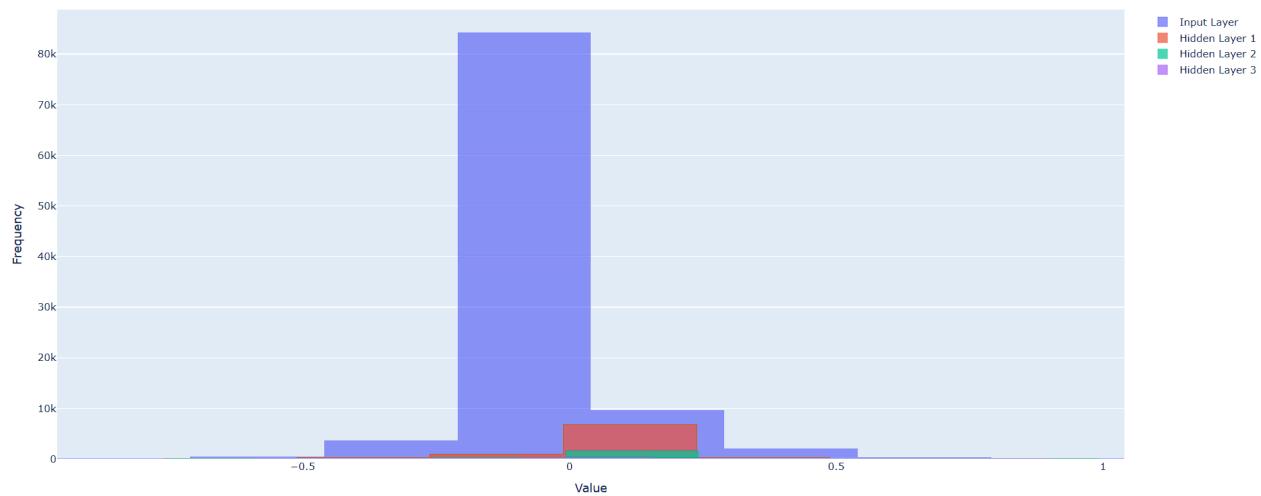


Berdasarkan grafik yang dihasilkan, keempat model tidak terlalu memiliki perbedaan yang signifikan, baik secara bentuk grafik maupun nilai Training Loss dan Validation Loss yang dihasilkan.

Distribusi bobot dari semua layer pada model pengujian berbagai konfigurasi fungsi aktivasi terurut sesuai dengan tabel diatas:

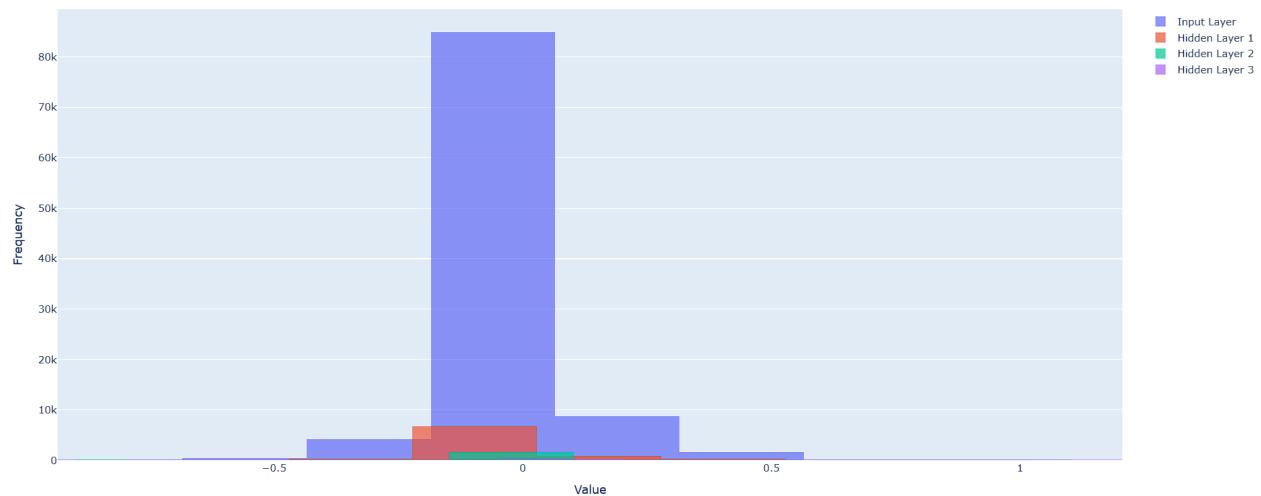
- Linear

Weight Distribution

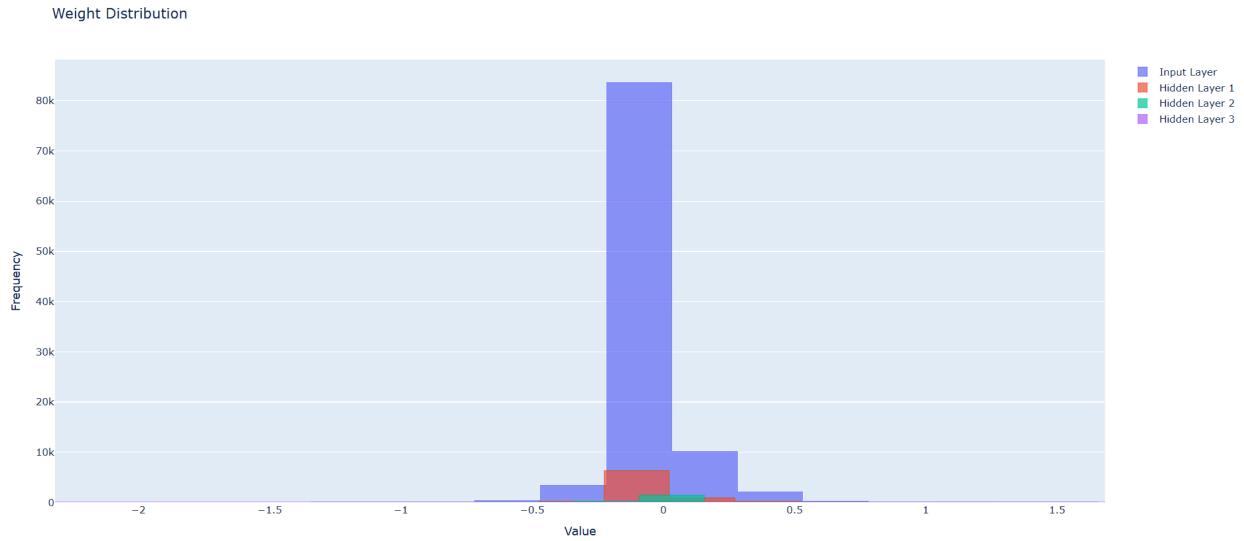


● ReLU

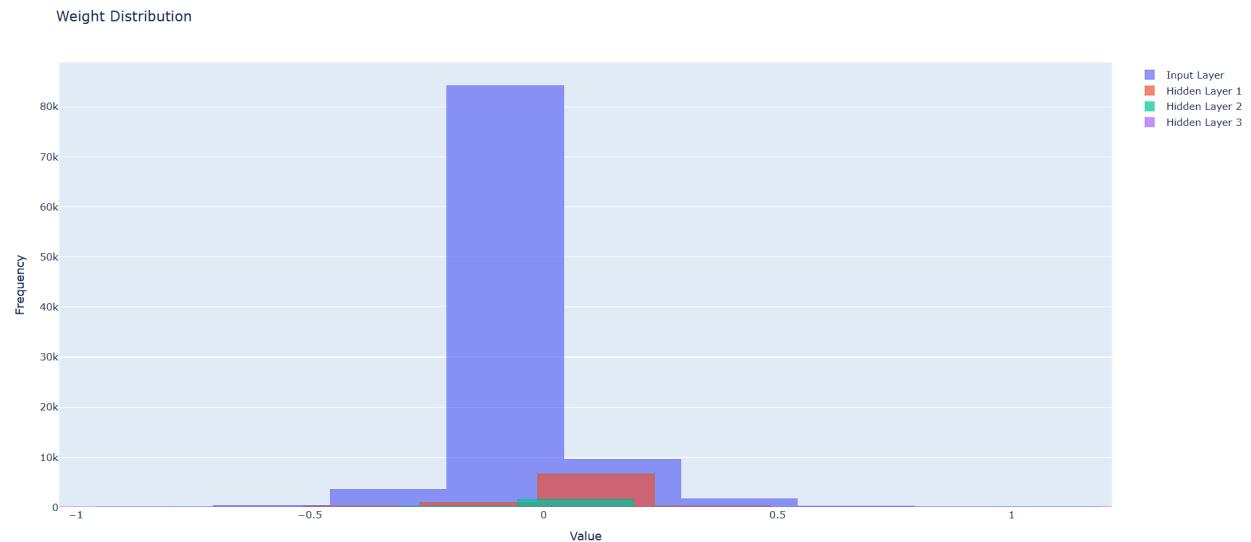
Weight Distribution



● Sigmoid



- Tanh

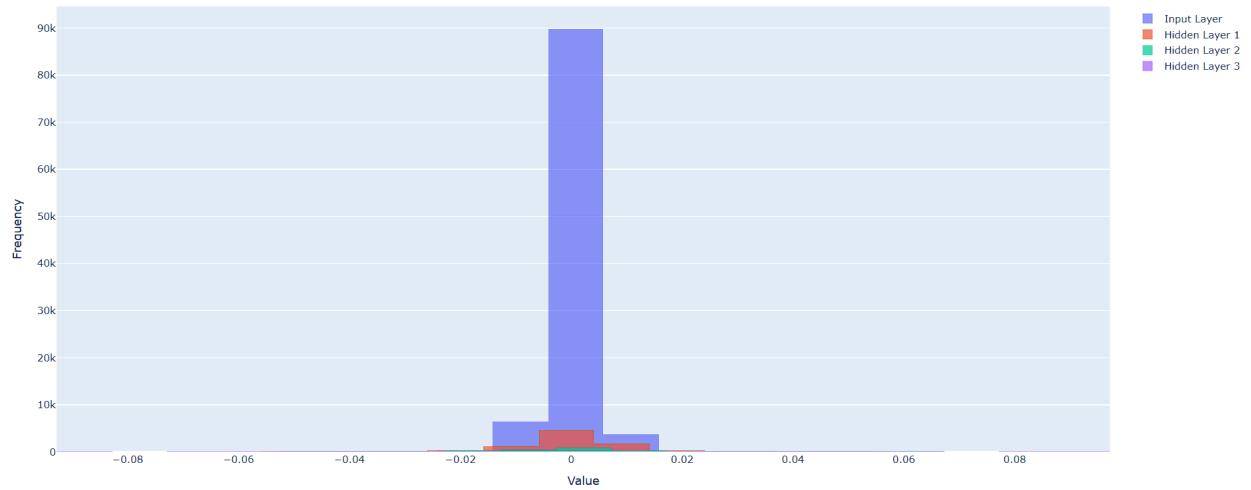


Berdasarkan keempat histogram yang dihasilkan, terlihat bahwa bobot input layer mendominasi dengan frekuensi bobot tertinggi dan terpusat di sekitar nol. Bobot hidden layer 1 memiliki distribusi yang lebih kecil dan sedikit bergeser dari nol. Bobot hidden layer 2 dan hidden layer 3 memiliki frekuensi yang lebih rendah dan hampir tidak terlihat pada seluruh grafik untuk hidden layer 3, yang menunjukkan bahwa bobot pada layer tersebut jumlahnya lebih sedikit atau memiliki distribusi yang lebih tersebar. Pola distribusi ini relatif konsisten di antara keempat konfigurasi fungsi aktivasi.

Distribusi gradient dari semua layer pada model pengujian berbagai konfigurasi fungsi aktivasi terurut sesuai dengan tabel diatas:

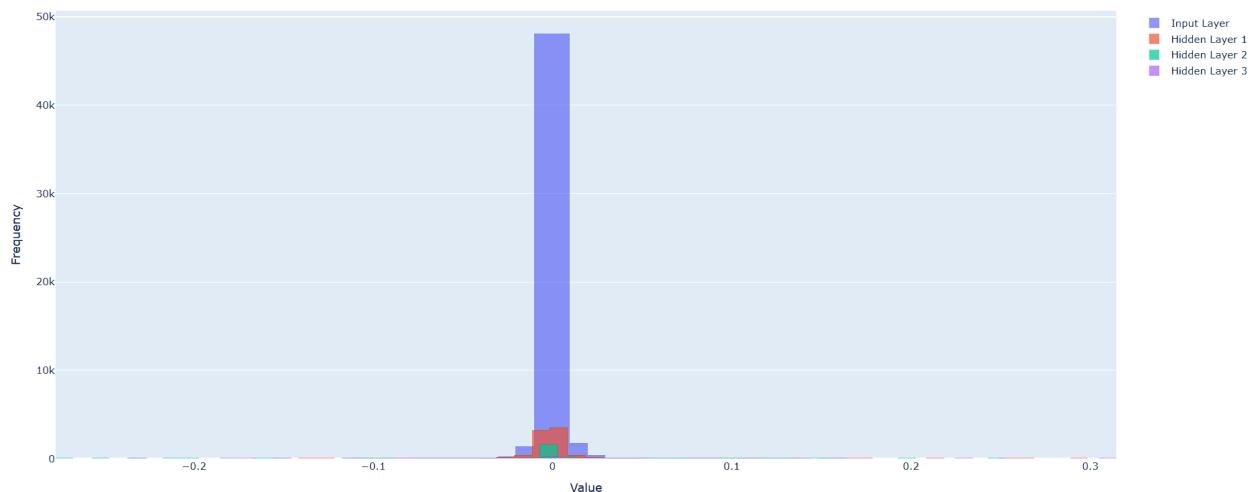
- Linear

Gradient Distribution

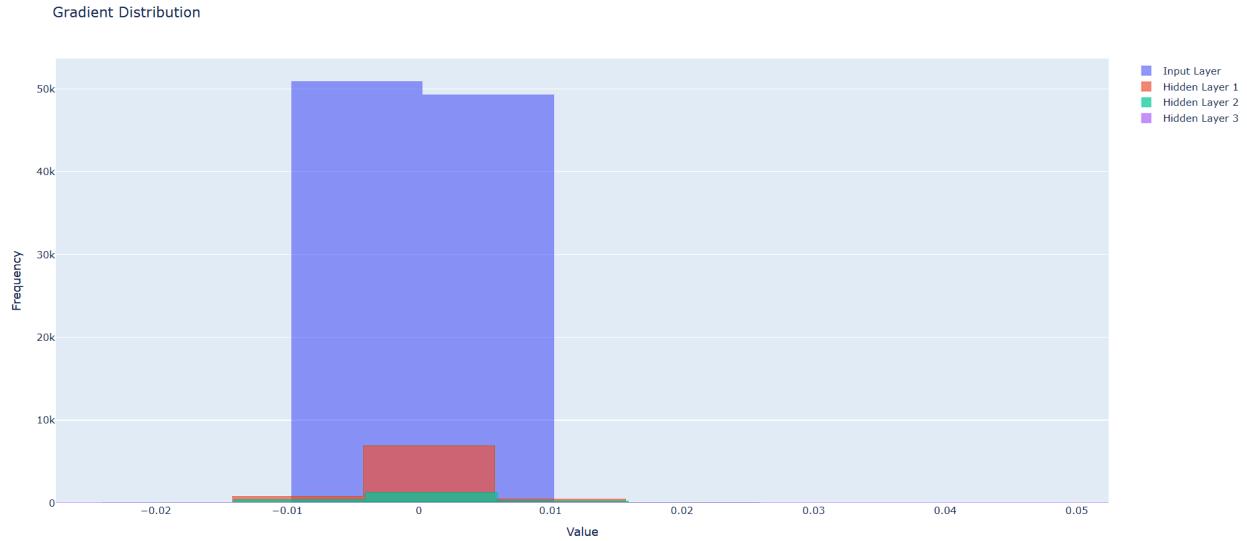


● ReLU

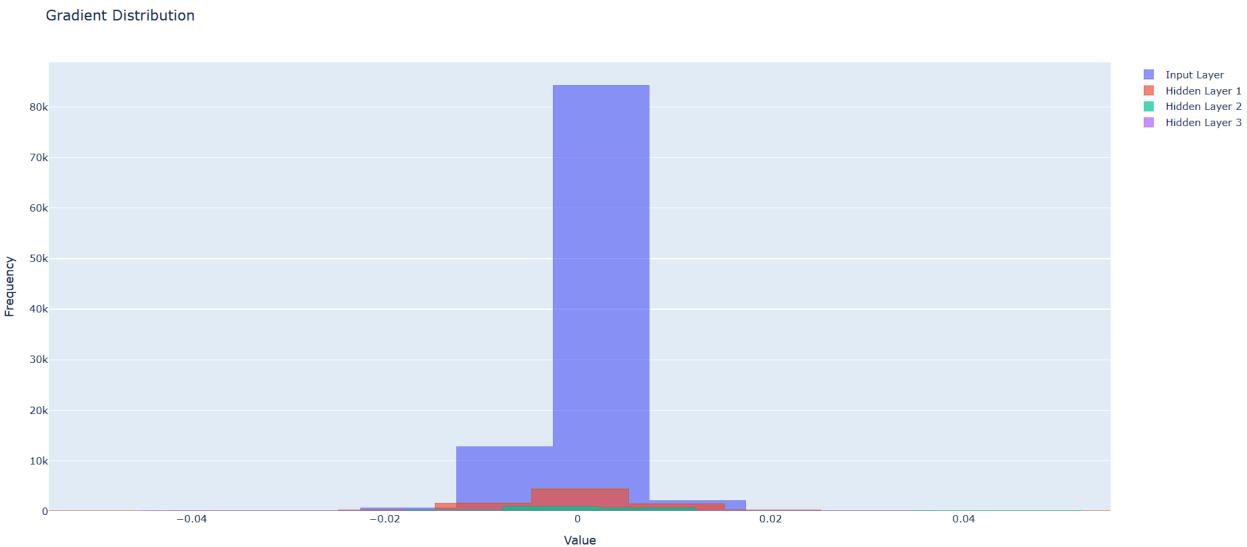
Gradient Distribution



● Sigmoid



- Tanh



Berdasarkan keempat histogram yang dihasilkan, terlihat bahwa fungsi aktivasi Sigmoid memiliki distribusi gradient yang sedikit lebih merata dan tersebar dibandingkan dengan fungsi aktivasi lainnya yang sangat terkonsentrasi di sekitar nilai nol dengan puncak yang sangat tinggi.

3.4 Pengaruh learning rate

Dari hasil pengujian dengan berbagai konfigurasi learning rate:

Learning Rate	Library	Implementasi
0.1	0.8449	0.8449
0.06	0.8449	0.8449
0.006	0.8449	0.8449

0.0009	0.8449	0.8449
--------	--------	--------

Hasil pengujian menunjukkan bahwa variasi learning rate tidak mempengaruhi akurasi model dalam eksperimen ini. Baik untuk learning rate 0.1, 0.01, 0.001, maupun 0.0001, model selalu mencapai akurasi sebesar 84.49% pada data uji. Hal ini terjadi baik pada implementasi manual FFNN maupun MLPClassifier dari Sklearn, yang menunjukkan bahwa algoritma pembelajaran bekerja secara konsisten terlepas dari perubahan nilai learning rate.

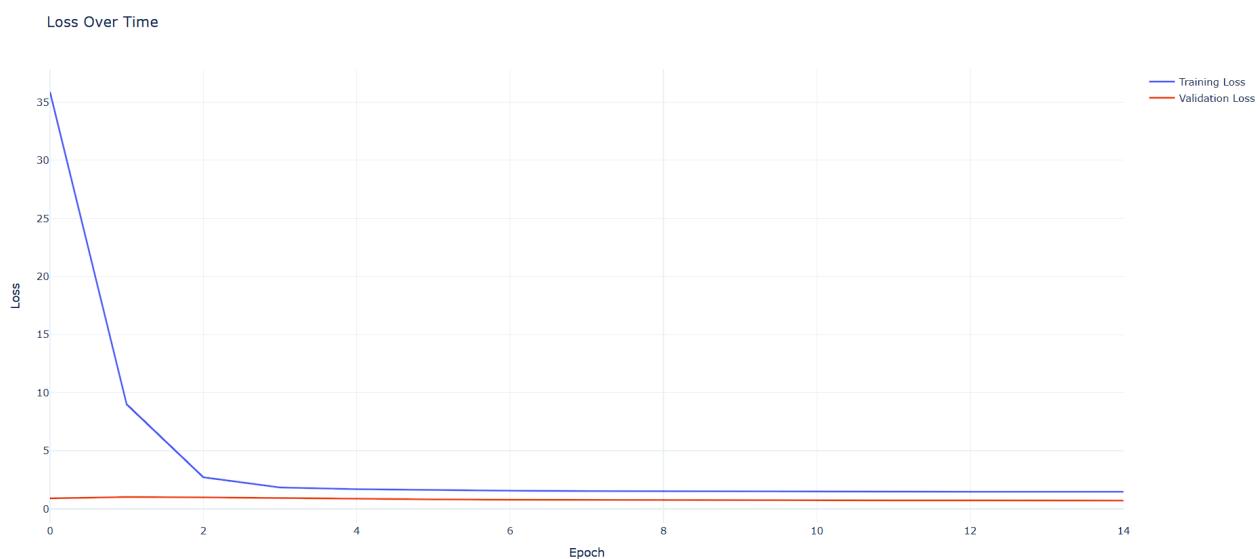
Secara teori, learning rate berpengaruh terhadap bagaimana model memperbarui bobotnya selama proses training. Learning rate yang terlalu besar dapat menyebabkan model melompat-lompat dan sulit mencapai konvergensi, sementara learning rate yang terlalu kecil dapat menyebabkan proses training menjadi sangat lambat. Namun, dalam eksperimen ini, hasil yang konstan mengindikasikan bahwa learning rate yang diuji masih berada dalam rentang yang aman sehingga tidak menghambat proses pembelajaran model.

Kemungkinan lain yang menyebabkan hasil ini adalah jumlah iterasi atau epoch yang kurang banyak, sehingga meskipun learning rate kecil, model tetap berhasil mencapai titik konvergensi yang sama.

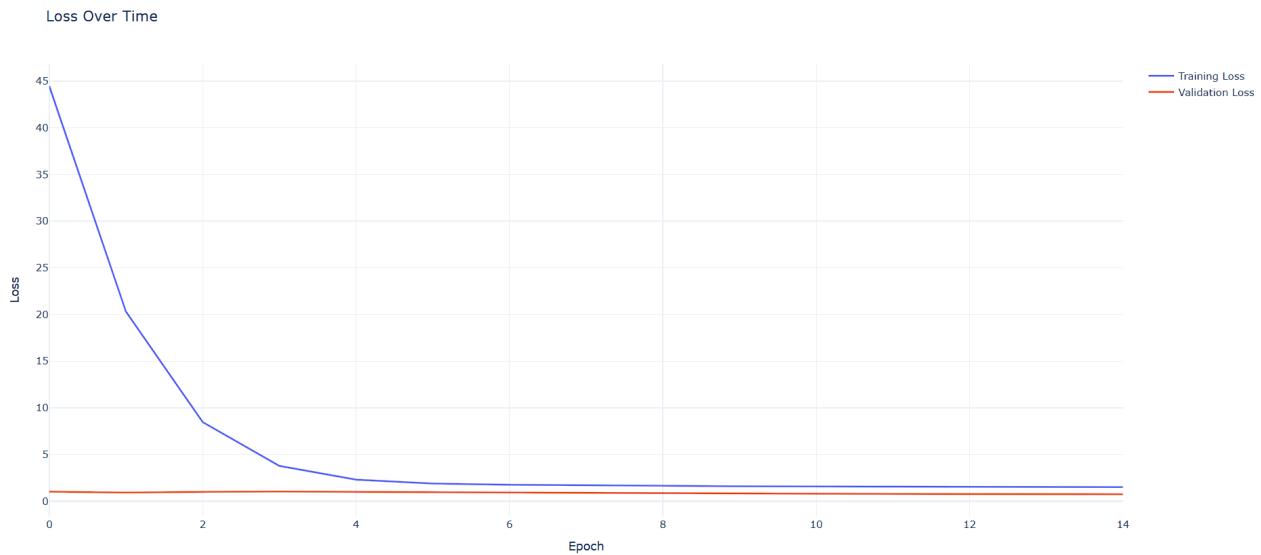
Dari hasil ini, dapat disimpulkan bahwa learning rate dalam rentang 0.0001 hingga 0.1 masih dapat menghasilkan hasil optimal untuk model ini, namun eksperimen lebih lanjut dapat dilakukan dengan learning rate yang lebih ekstrem (misalnya 1.0 atau 0.00001) untuk melihat apakah ada perubahan signifikan dalam performa model. Selain itu, dapat diuji apakah perubahan learning rate berpengaruh terhadap kecepatan konvergensi model, bukan hanya akurasi akhirnya.

Grafik loss pengujian berbagai konfigurasi learning rate terurut sesuai dengan tabel diatas:

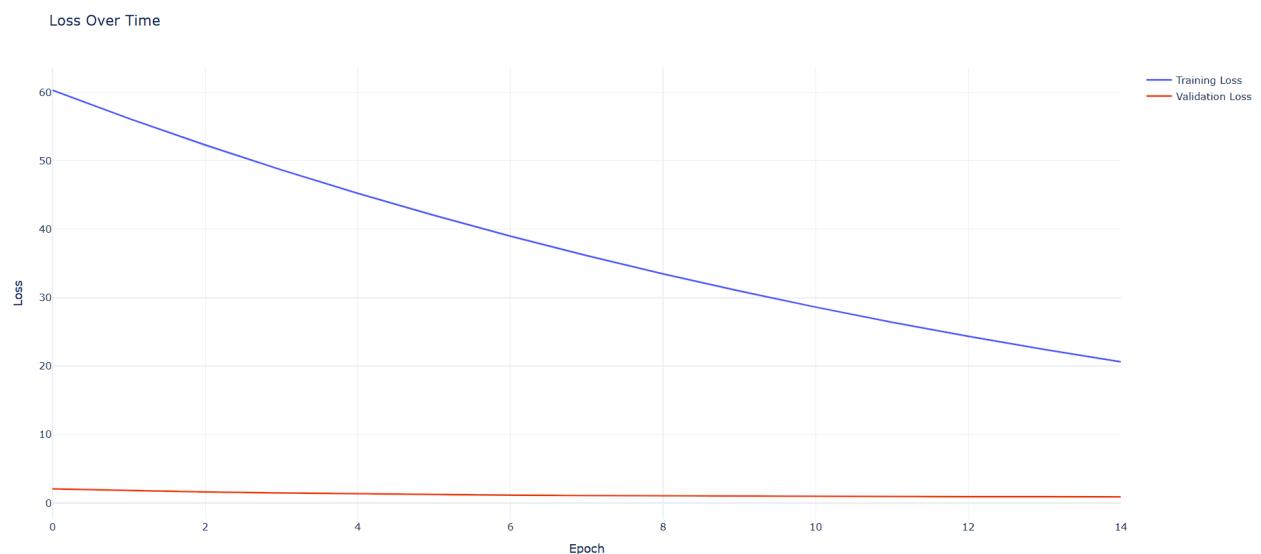
- Learning Rate 0.1



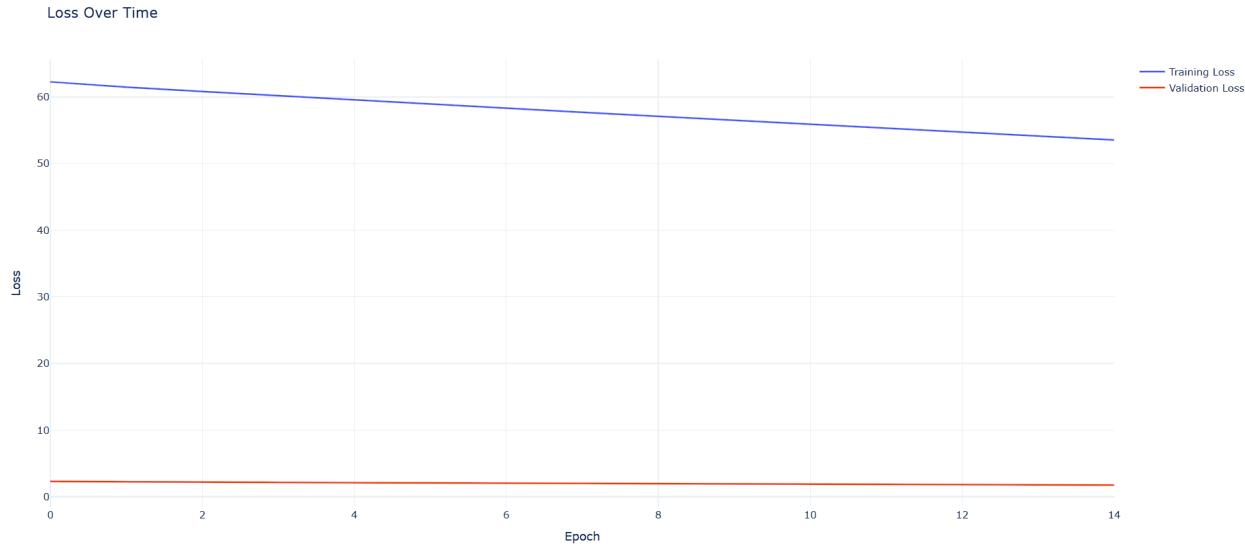
- Learning Rate 0.06



- Learning Rate 0.006



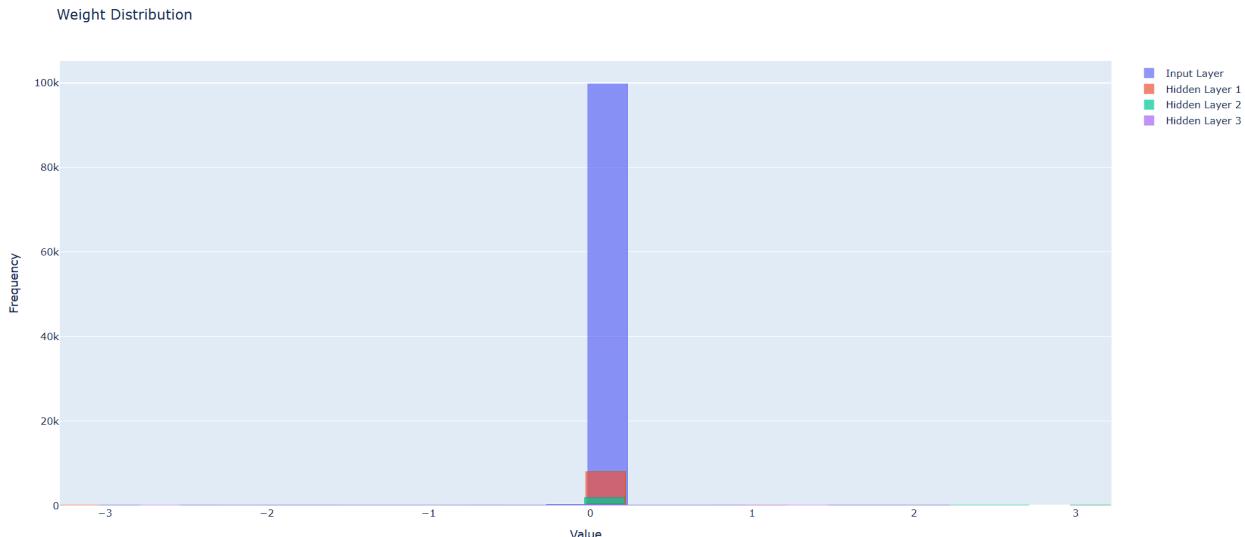
- Learning Rate 0.0009



Berdasarkan grafik yang dihasilkan, terlihat bahwa learning rate sangat mempengaruhi kecepatan dan efektivitas pembelajaran model. Learning rate yang lebih tinggi menunjukkan penurunan Training Loss yang sangat cepat dan drastis dalam epoch-epoch awal, sedangkan learning rate yang lebih rendah menunjukkan penurunan training loss yang sangat lambat dan semakin linier. Validation Loss tetap rendah dan stabil untuk semua konfigurasi, tetapi model dengan learning rate lebih tinggi mencapai keseimbangan yang lebih baik antara Training dan Validation Loss, mengindikasikan pembelajaran yang efisien tanpa overfitting yang signifikan.

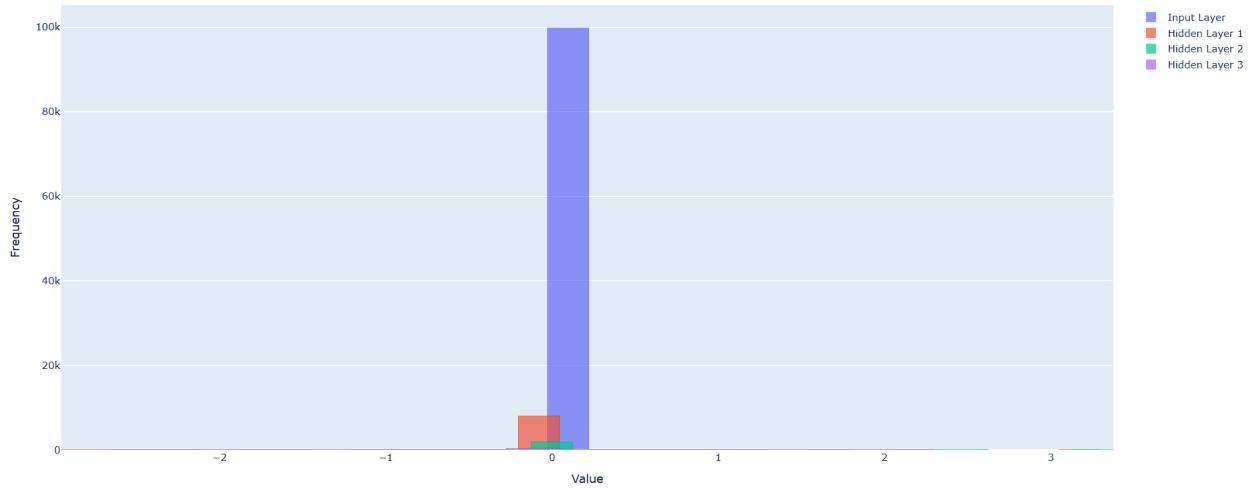
Distribusi bobot dari semua layer pada model pengujian berbagai konfigurasi learning rate terurut sesuai dengan tabel diatas:

- Learning Rate 0.1



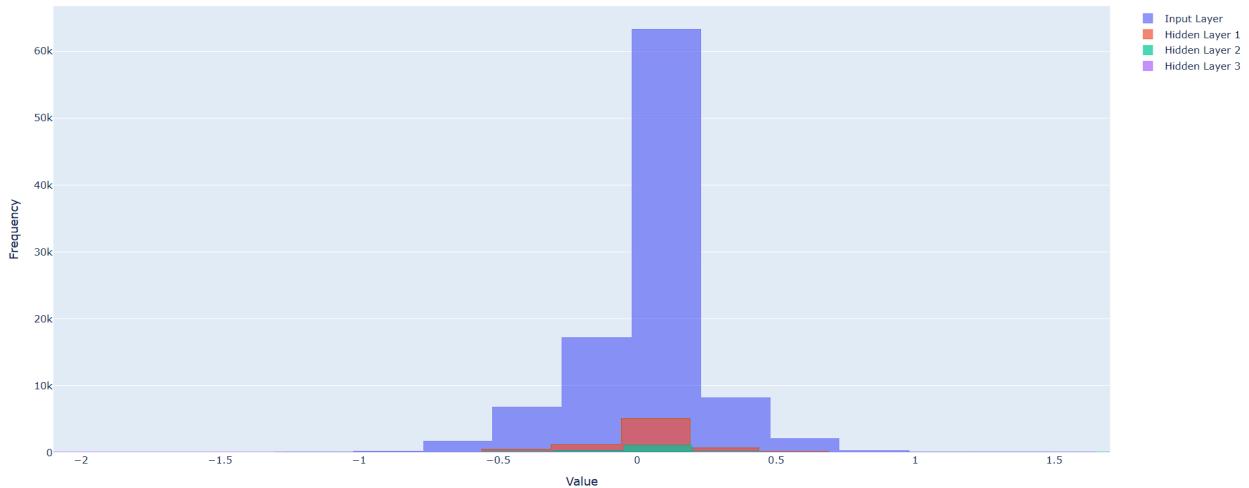
- Learning Rate 0.06

Weight Distribution

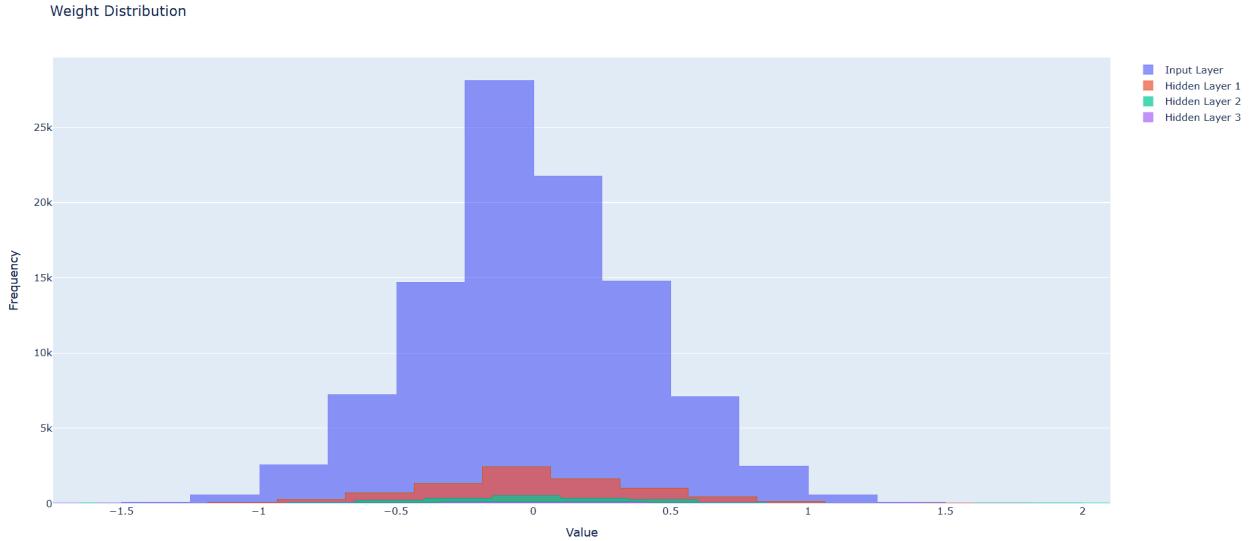


- Learning Rate 0.006

Weight Distribution



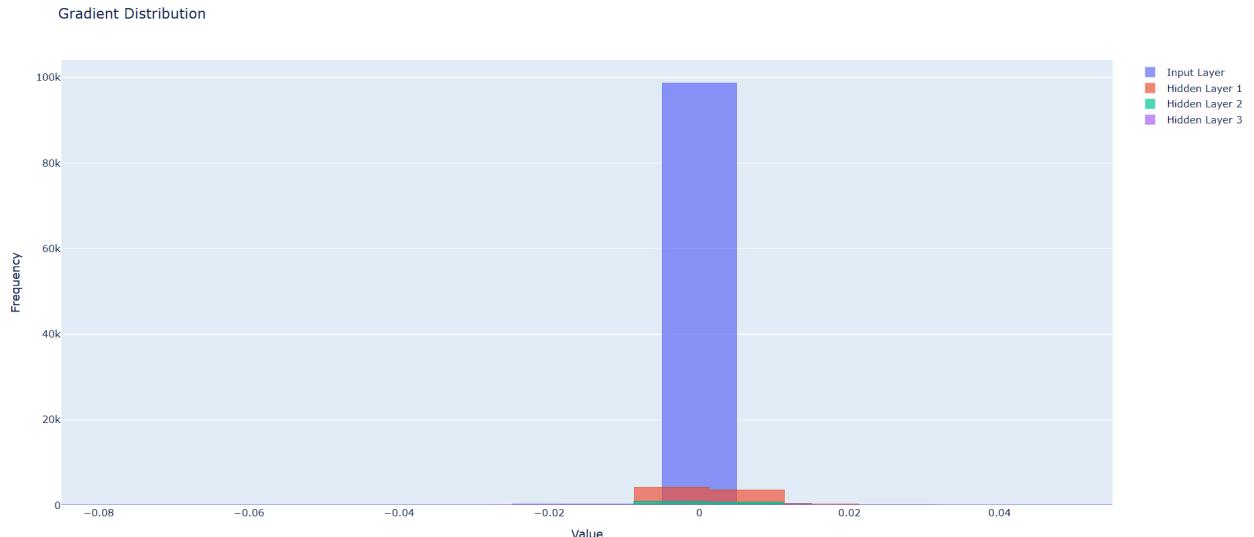
- Learning Rate 0.0009



Berdasarkan keempat histogram yang dihasilkan, terlihat bahwa semakin kecil learning rate, semakin tersebar distribusi bobot dan semakin banyak variasi nilai bobot di seluruh layer jaringan. Learning rate yang lebih besar cenderung menghasilkan bobot yang sangat terkonsentrasi di sekitar nol, sementara learning rate yang lebih kecil memungkinkan eksplorasi ruang parameter yang lebih luas.

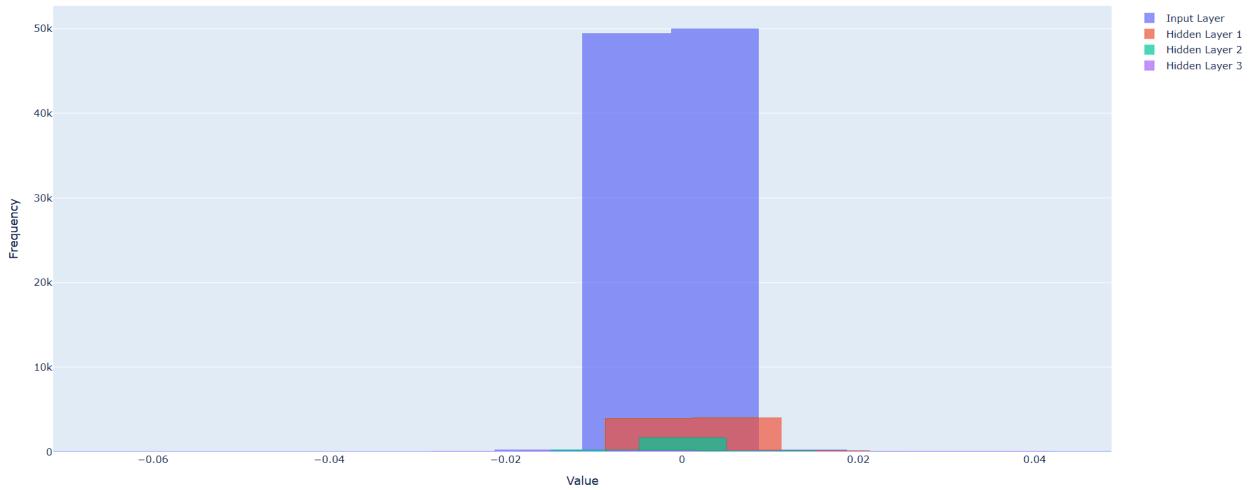
Distribusi gradient dari semua layer pada model pengujian berbagai konfigurasi learning rate terurut sesuai dengan tabel diatas:

- Learning Rate 0.1



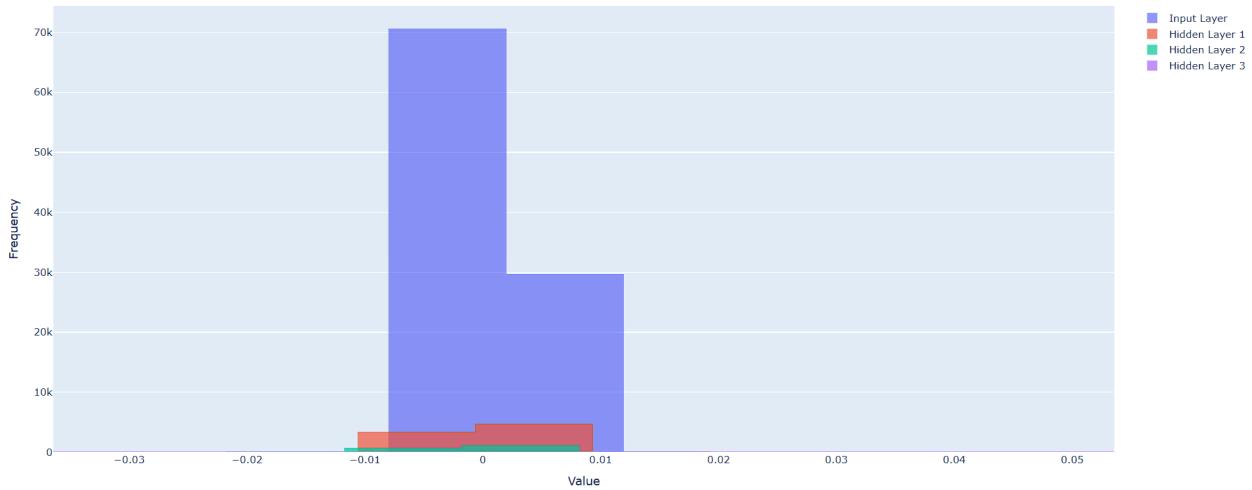
- Learning Rate 0.06

Gradient Distribution

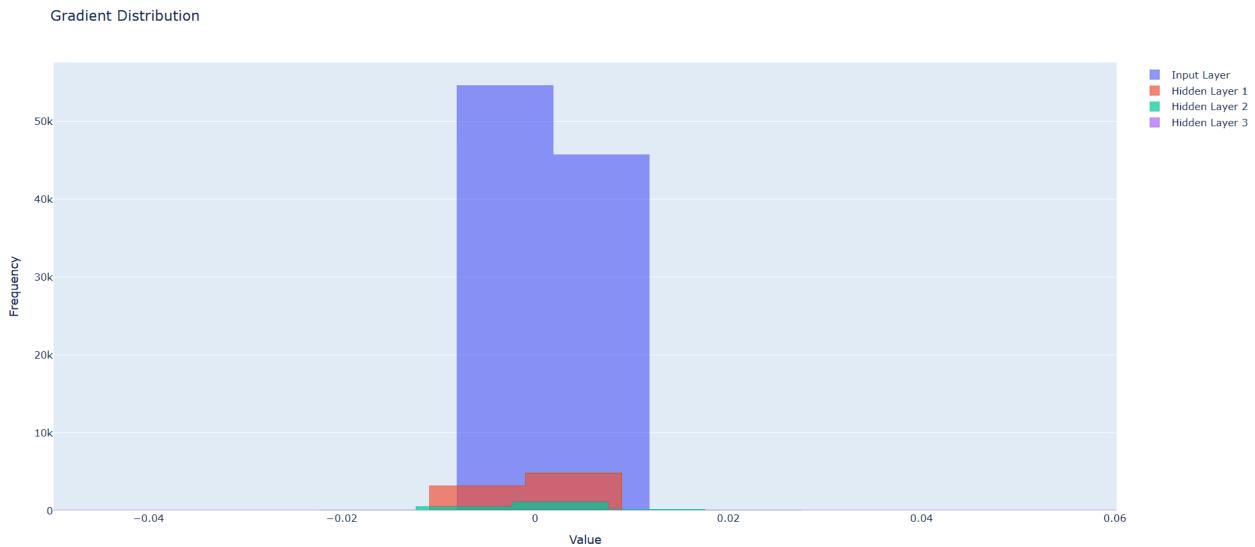


- Learning Rate 0.006

Gradient Distribution



- Learning Rate 0.0009



Berdasarkan keempat histogram yang dihasilkan, terlihat bahwa distribusi gradient pada learning rate yang lebih kecil menghasilkan distribusi yang lebih merata dibandingkan dengan learning rate yang lebih tinggi.

3.5 Pengaruh Inisialisasi Bobot

Iterasi maksimum yang digunakan dalam mengukur performa konfigurasi bobot bernilai lebih kecil (15) dibandingkan default parameter. Hal ini dikarenakan pada versi library, inisialisasinya sudah konvergen lebih dahulu sebelum versi implementasi menyentuh iterasi maksimum.

Dari hasil pengujian dengan berbagai konfigurasi inisialisasi bobot:

Metode Inisialisasi	Library	Implementasi
Normal	0.7144	0.7144
Zero	0,1142	0,1142
Uniform	0,1142	0,1142
Normal (Xavier)	0,6585	0,6585
Uniform (Xavier)	0,6671	0,6671
Normal (He)	0,7478	0,7478
Uniform (He)	0,7675	0,7675

Hasil pengujian terbaru menunjukkan pola yang konsisten dalam performa model MLPClassifier berdasarkan metode inisialisasi bobot yang digunakan. Pada inisialisasi normal standar, kedua implementasi mencapai akurasi identik sebesar 0.7144, yang mengindikasikan implementasi dari scratch berhasil mereplikasi metode inisialisasi ini dengan tepat.

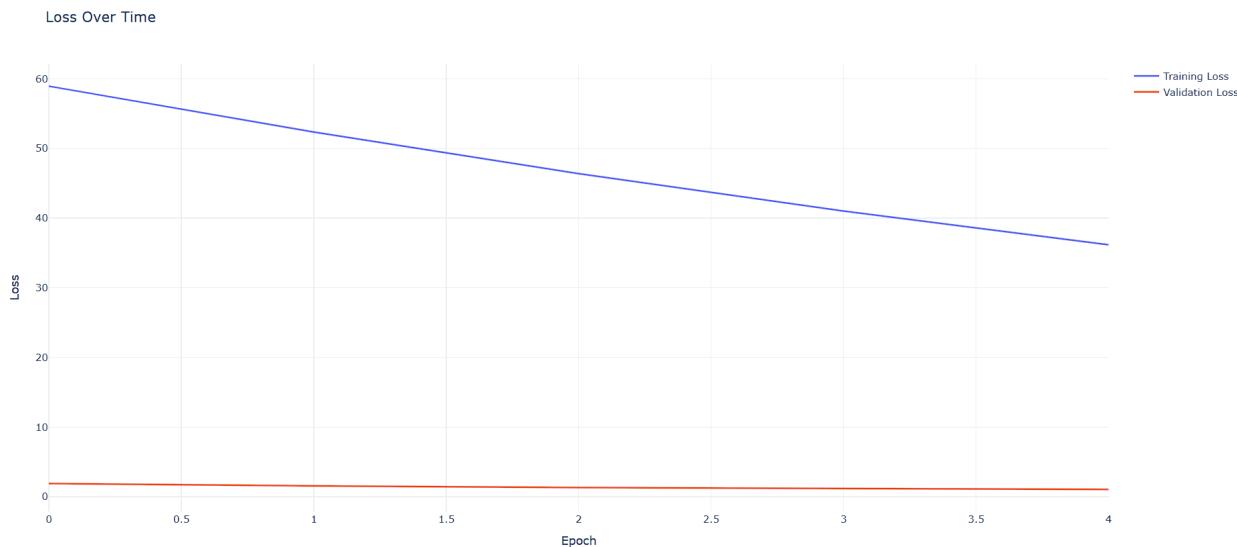
Inisialisasi zero dan uniform menunjukkan performa yang sangat rendah dengan akurasi hanya 0.1142 pada kedua implementasi. Konsistensi hasil ini menegaskan bahwa inisialisasi yang tidak tepat dapat menyebabkan model gagal konvergen ke solusi yang optimal, kemungkinan karena masalah simetri atau vanishing gradient.

Menariknya, dalam dataset ini, inisialisasi normal dengan scaling xavier menghasilkan performa yang lebih rendah daripada normal standar, dengan akurasi 0.6585 pada kedua implementasi. Sementara itu, metode uniform dengan scaling xavier sedikit lebih baik dengan akurasi 0.6671, meskipun masih di bawah metode normal standar. Hasil ini menunjukkan bahwa scaling xavier tidak selalu optimal untuk semua dataset atau arsitektur jaringan.

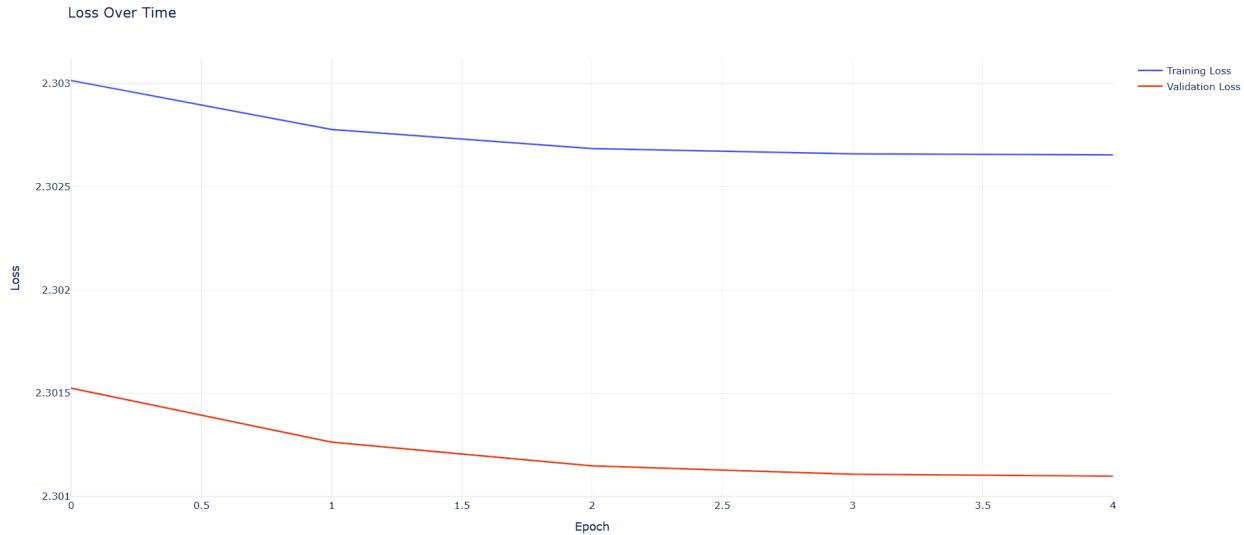
Performa terbaik dalam pengujian ini dicapai dengan metode inisialisasi uniform he, yang menghasilkan akurasi 0.7675 pada kedua implementasi. Inisialisasi normal he juga menunjukkan hasil yang cukup baik dengan akurasi 0.7478, meskipun di bawah uniform he.

Grafik loss pengujian berbagai konfigurasi inisialisasi bobot terurut sesuai dengan tabel diatas:

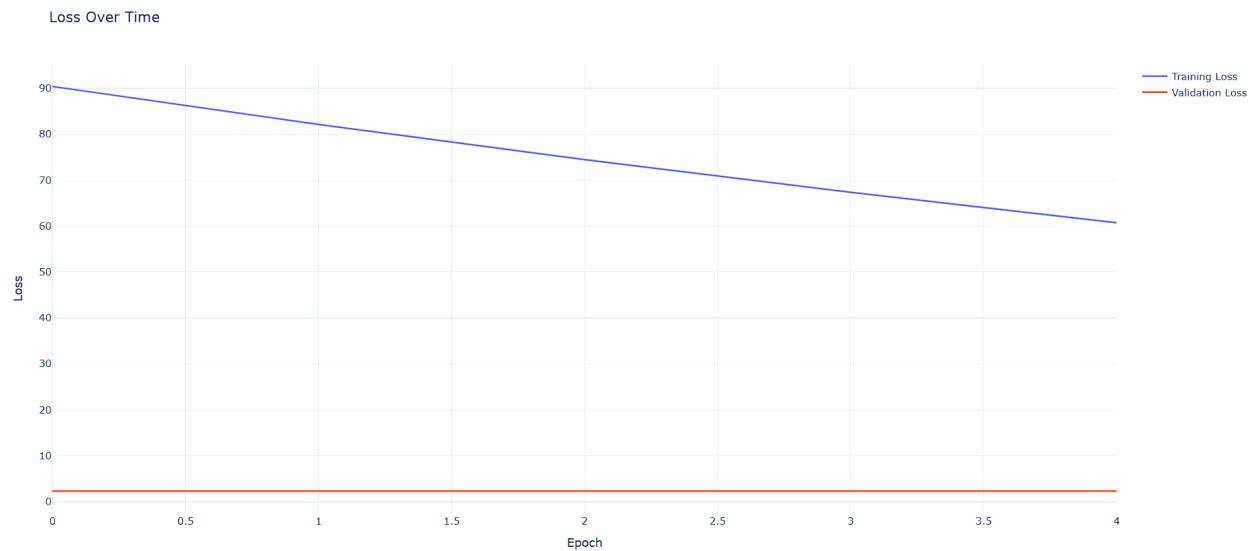
- Normal



- Zero



- Uniform

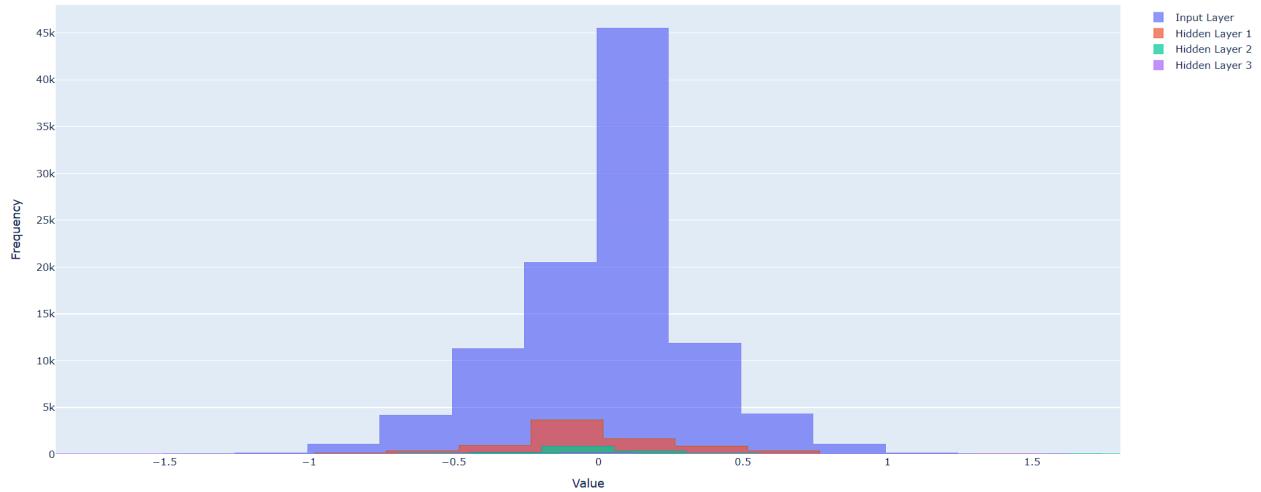


Berdasarkan grafik yang dihasilkan, terlihat perbedaan signifikan dalam pola pembelajaran model, inisialisasi Normal dan Uniform menunjukkan penurunan Training Loss yang lebih konsisten dan bertahap, sementara inisialisasi Zero menghasilkan grafik dengan skala loss berbeda dan menunjukkan penurunan Validation Loss yang tidak terlihat pada metode lainnya.

Distribusi bobot dari semua layer pada model pengujian berbagai konfigurasi inisialisasi bobot terurut sesuai dengan tabel diatas:

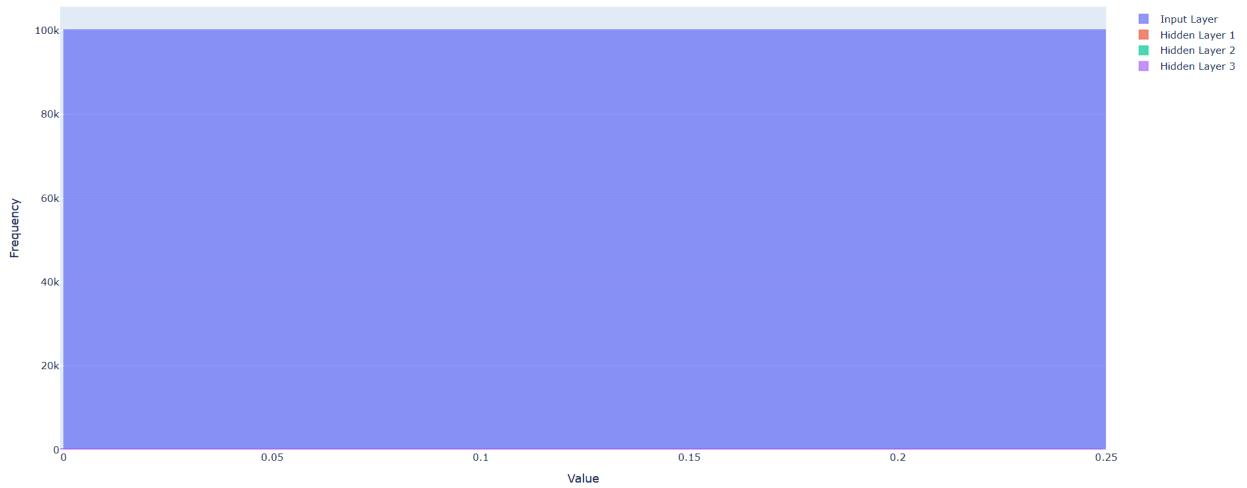
- Normal

Weight Distribution

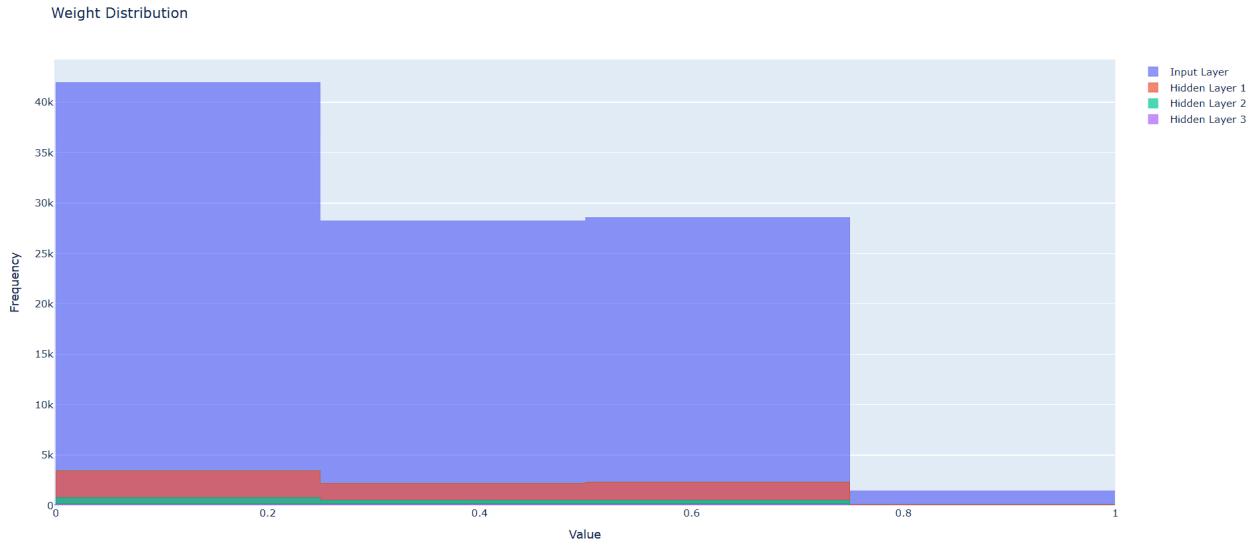


● Zero

Weight Distribution



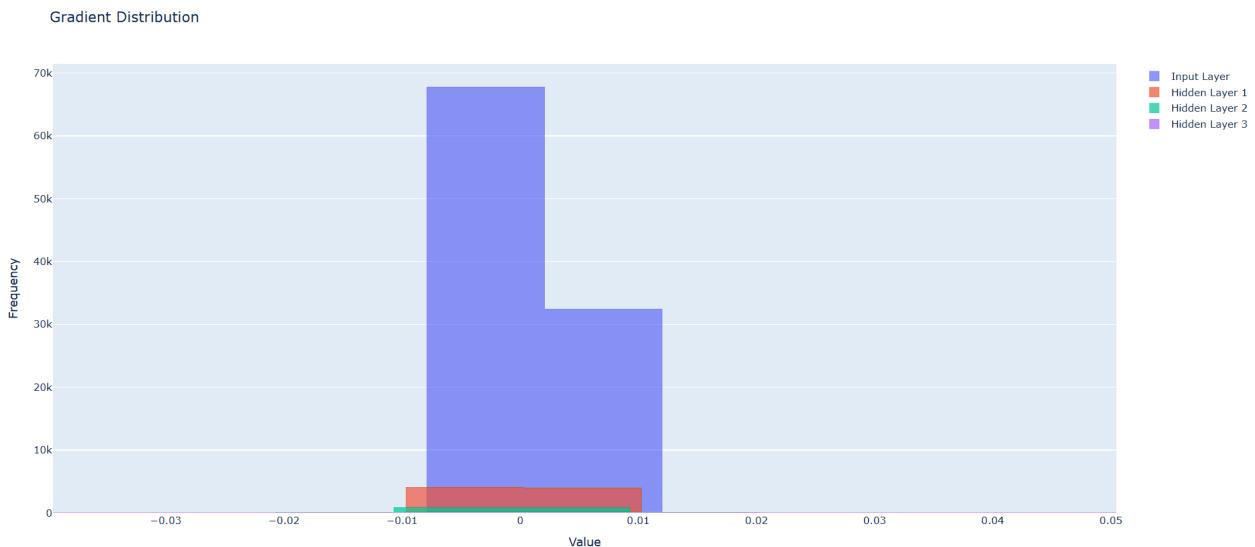
● Uniform



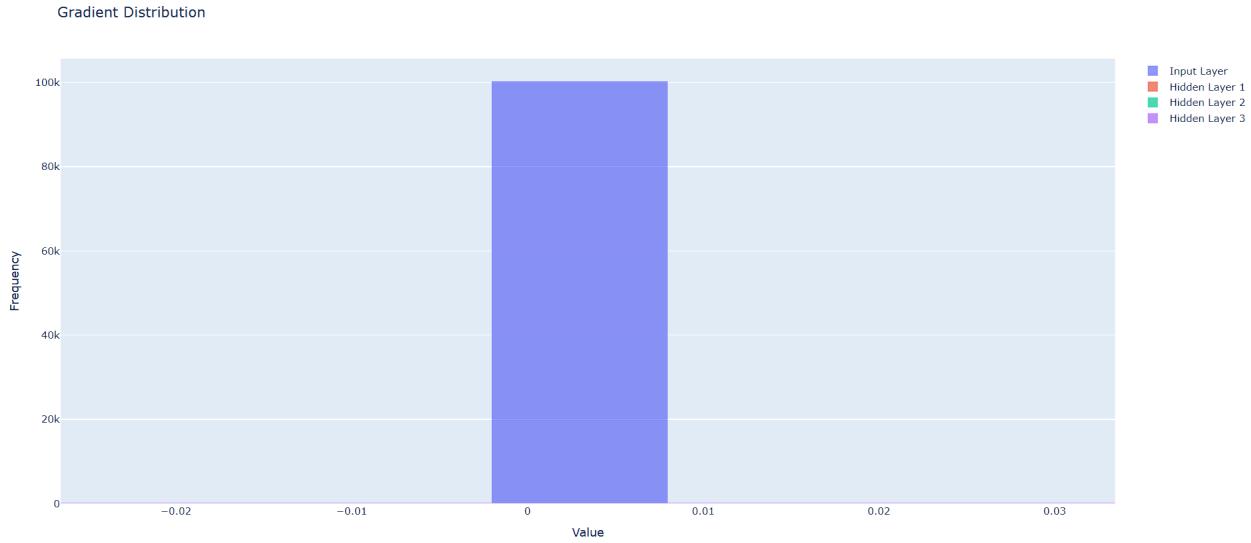
Berdasarkan ketiga histogram yang dihasilkan, terlihat bahwa distribusi yang dihasilkan sesuai dengan nama metode yang digunakan, seperti metode normal menghasilkan distribusi bobot yang normal dan metode uniform menghasilkan distribusi bobot yang uniform dan berada pada range lower bound dan upper bound yang digunakan, sedangkan pada metode zero menghasilkan bobot yang berpusat di 0 karena metode tersebut menghasilkan seluruh inisialisasi bobot bernilai 0 terlihat pada persebaran bobot input layernya dan persebaran layer pada layer-layer selanjutnya juga mendekati 0.

Distribusi gradient dari semua layer pada model pengujian berbagai konfigurasi inisialisasi bobot terurut sesuai dengan tabel diatas:

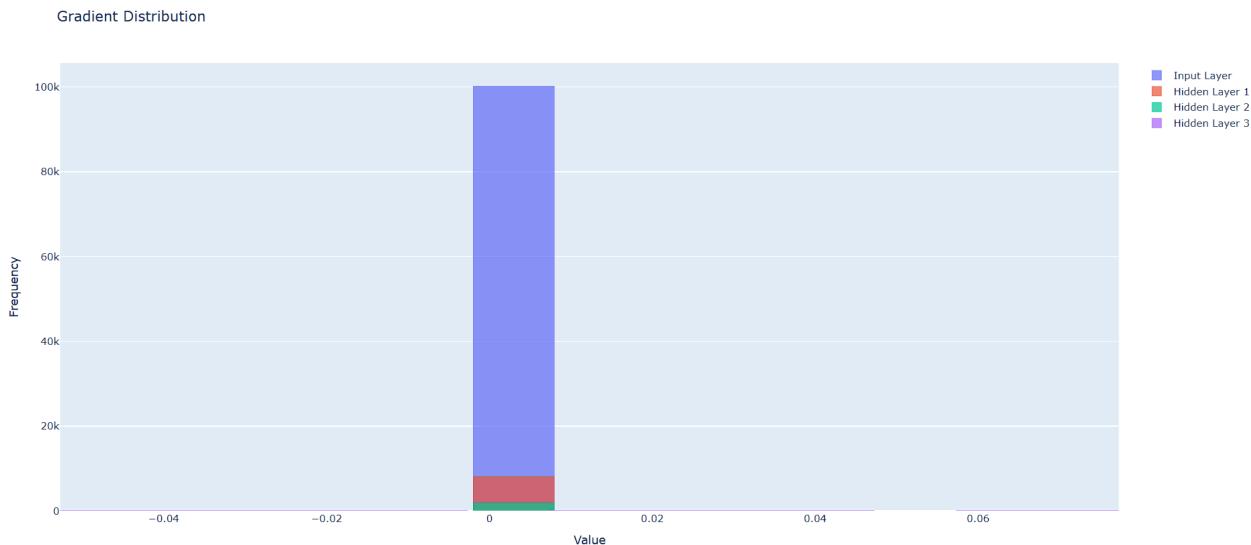
- Normal



- Zero



- Uniform



Berdasarkan ketiga histogram yang dihasilkan, terlihat bahwa metode normal memiliki distribusi nilai gradien yang relatif tersebar dengan dua puncak utama, sedangkan pada metode zero distribusi sangat terkonsentrasi pada satu titik nilai dan pada metode uniform distribusi sangat sangat sempit serta masih cenderung kurang bervariasi dibandingkan dengan inisialisasi Normal.

3.6 Pengaruh Regulariasasi

Dari hasil pengujian dengan berbagai konfigurasi regularisasai:

L1	Library	Implementasi
0,1	0,845	0,845
0,01	0,8475	0,8475

0,001	0,8450	0,8450
0,0001	0.8449	0.8449
0,00001	0.8449	0.8449

L2	Library	Implementasi
0,1	0.86	0.86
0,01	0,846	0,846
0,001	0,8447	0,8447
0,0001	0.8449	0.8449
0,00001	0.8449	0.8449

Hasil pengujian menunjukkan pengaruh signifikan dari parameter regularisasi L1 dan L2 terhadap performa model MLPClassifier. Pada regularisasi L1, nilai parameter 0.01 memberikan hasil terbaik dengan akurasi 0.8475 pada kedua implementasi, menunjukkan peningkatan dibanding nilai parameter yang lebih besar (0.1) maupun yang lebih kecil. Nilai parameter L1 sebesar 0.1 menghasilkan akurasi 0.845, sementara nilai parameter yang lebih kecil dari 0.01 menunjukkan penurunan performa secara bertahap, dengan 0.001 menghasilkan akurasi 0.8450 dan nilai 0.0001 serta 0.00001 menghasilkan akurasi 0.8449.

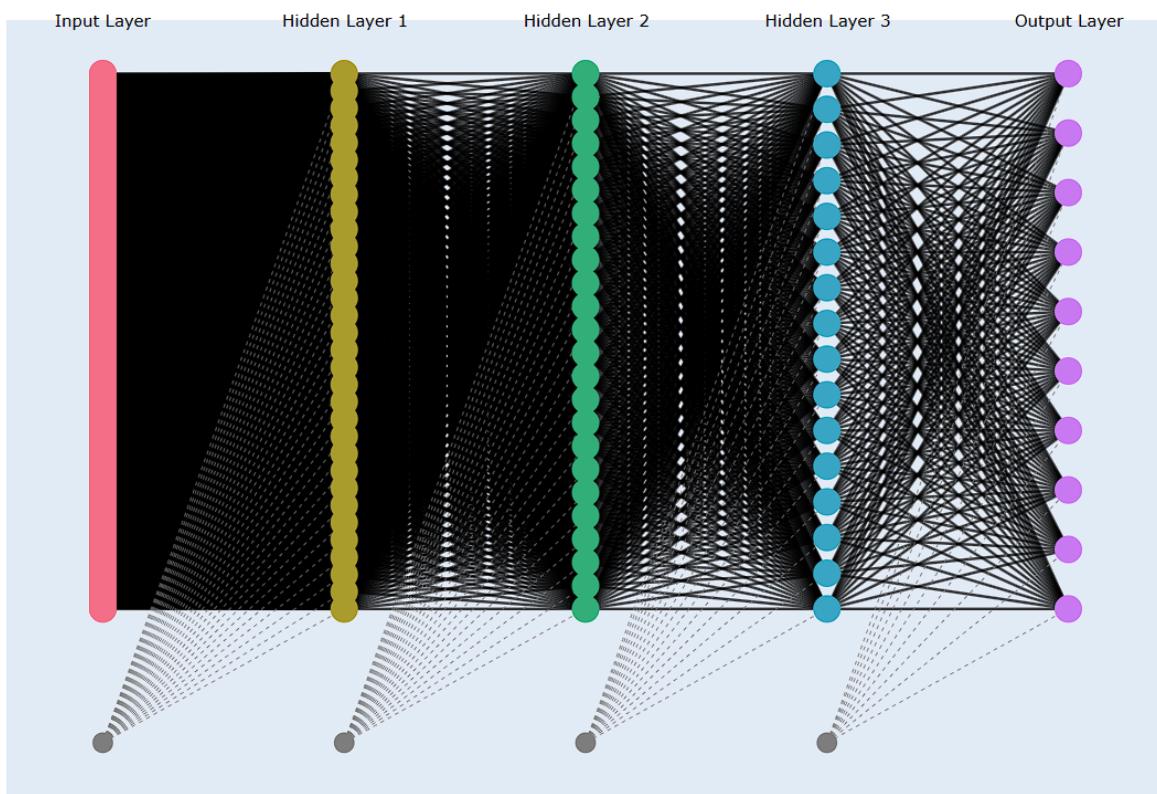
Untuk regularisasi L2, pola yang berbeda terlihat ketika nilai parameter terbesar (0.1) justru memberikan performa terbaik dengan akurasi 0.86 pada kedua implementasi. Terdapat penurunan performa yang konsisten seiring dengan pengurangan nilai parameter, parameter 0.01 menghasilkan akurasi 0.846, parameter 0.001 menghasilkan 0.8447, dan parameter 0.0001 serta 0.00001 keduanya menghasilkan akurasi 0.8449.

Perbandingan kedua jenis regularisasi menunjukkan bahwa regularisasi L2 dengan parameter 0.1 memberikan hasil terbaik secara keseluruhan dengan akurasi 0.86. Hal ini mengindikasikan bahwa untuk dataset yang digunakan, pendekatan L2 yang meminimalkan magnitude bobot secara keseluruhan lebih efektif dibandingkan pendekatan L1 yang mendorong sparsitas. Namun, penting juga untuk dicatat bahwa efektivitas regularisasi L2 sangat bergantung pada nilai parameter yang tepat, karena performa menurun secara signifikan ketika parameter dikurangi.

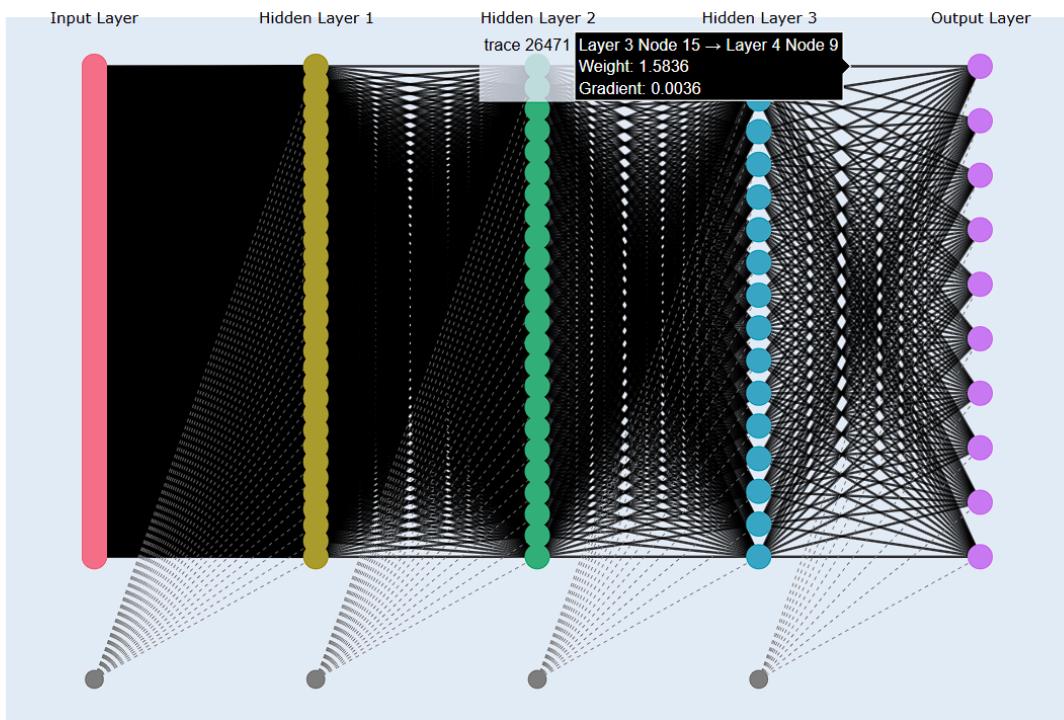
4. Visualisasi

4.1 Struktur Jaringan

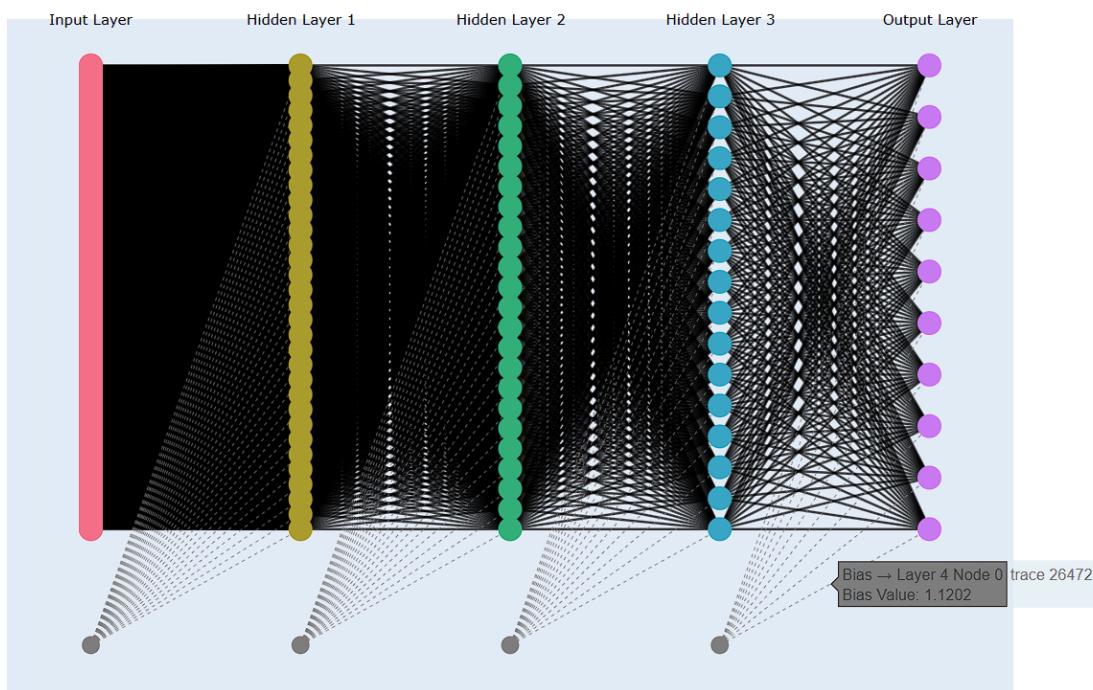
Neural Network Structure with Bias Nodes



Neural Network Structure with Bias Nodes



Neural Network Structure with Bias Nodes



Layer Size = [784, 32, 24, 16, 10]

C. Kesimpulan dan Saran

Klasifikasi suatu gambar menjadi angka dapat dilakukan dengan memanfaatkan Neural Network terutama Multilayer Perceptron seperti Feed Forward Neural Network. Feed Forward Neural Network sendiri dapat mengklasifikasi angka dengan sangat akurat yaitu dengan akurasi mencapai lebih dari 90%.

D. Pembagian Tugas

NIM	Nama	Tugas
13522014	Raden Rafly Hanggaraksa Budiarto	<ol style="list-style-type: none">1. Metode inisialisasi bobot2. Perbandingan performa model dengan library3. Menyusun laporan
13522084	Dhafin Fawwaz Ikramullah	<ol style="list-style-type: none">1. Forward propagation2. Backward propagation3. Memastikan output sama dengan MLPClassifier sklearn.4. Membuat runner main dengan berbagai argumentnya5. Menyusun laporan
13522092	Sa'ad Abdul Hakim	<ol style="list-style-type: none">1. Visualisasi2. Menyusun laporan

E. Referensi

- https://edunexcontentprodhot.blob.core.windows.net/edunex/nullfile/1741577891298_IF3270-M_gg03-FFNN-print?sv=2024-11-04&spr=https&st=2025-03-10T03%3A32%3A33Z&se=2027-03-10T03%3A32%3A33Z&sr=b&sp=r&sig=KdLqCzzo5Z3lQwTxt%2BEoNGBpLkPPB2YMdYxbAAqYzQ%3D&rsct=application%2Fpdf
- <https://arxiv.org/abs/1811.03378v1>
- <https://www.jeremyjordan.me/intro-to-neural-networks/>
- <https://www.jasonosajima.com/forwardprop>
- <https://www.jasonosajima.com/backprop>
- <https://numpy.org/doc/2.2/>
- https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html
- [https://math.libretexts.org/Bookshelves/Calculus/Calculus_\(OpenStax\)/14%3A_Differentiation_of_Functions_of_Several_Variables/14.05%3A_The_Chain_Rule_for_Multivariable_Functions](https://math.libretexts.org/Bookshelves/Calculus/Calculus_(OpenStax)/14%3A_Differentiation_of_Functions_of_Several_Variables/14.05%3A_The_Chain_Rule_for_Multivariable_Functions)
- <https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/>
- <https://douglasorr.github.io/2021-11-autodiff/article.html>
- https://www.cs.toronto.edu/~rgrosse/courses/csc2541_2022/tutorials/tut01.pdf