TUGAS BESAR 1

IF3270 PEMBELAJARAN MESIN

"Feedforward Neural Network"

Dosen:

Dr. Fariska Zakhralativa Ruskanda, S.T., M.T.

Kelompok 61:

13522140 Yasmin Farisah Salma

13522154 Chelvadinda ???

13522161 Mohammad Akmal Ramadan

IF3270 PEMBELAJARAN MESIN

PROGRAM STUDI TEKNIK INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

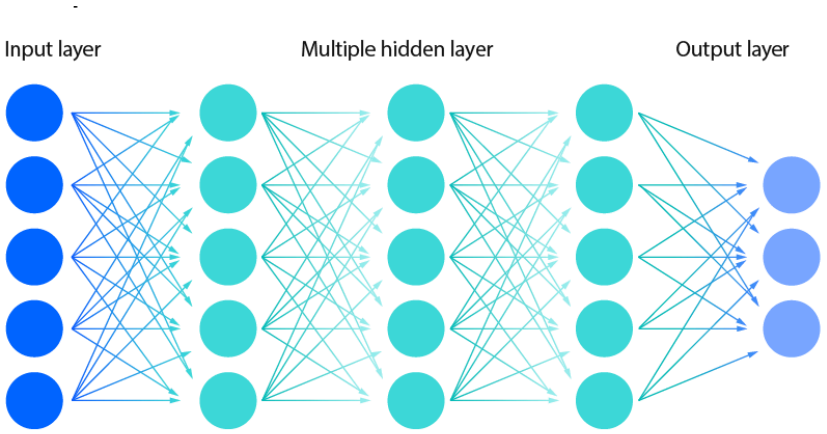
SEMESTER 1 TAHUN 2024/2025

DAFTAR ISI

BAB I
PENDAHULUAN

1.1 Deskripsi Persoalan

Tugas besar ini mengimplementasikan Feedforward Neural Network (FFNN) *from scratch* menggunakan bahasa pemrograman Python. Implementasi ini bertujuan untuk memberikan wawasan mendalam mengenai cara kerja *artificial neural network*, termasuk proses *forward propagation, backward propagation*, dan *weight update* dengan metode *gradient descent*. FFNN merupakan salah satu arsitektur *artificial neural network* yang terdiri atas beberapa layer neuron, yaitu satu *input layer*, satu atau lebih *hidden layer*, dan satu *output layer*. Setiap neuron dalam suatu layer terhubung dengan neuron di layer berikutnya melalui *weight* yang harus diinisialisasi dan diperbarui selama pelatihan model.



Dalam tugas ini, kami mengimplementasikan berbagai metode inisialisasi bobot. Selain itu, FFNN yang kami implementasikan juga dapat menerima parameter yang menentukan jumlah neuron di setiap lapisan serta berbagai fungsi aktivasi, termasuk Linear, ReLU, Sigmoid, Hyperbolic Tangent (Tanh), dan Softmax. Model juga mampu mengoptimalkan bobot berdasarkan fungsi *loss* yang telah ditentukan, yaitu Mean Squared Error (MSE), Binary Cross-Entropy, dan Categorical Cross-Entropy. Untuk menguji keandalan model, kami juga melakukan eksperimen dan analisis terhadap berbagai *hyperparameter*, termasuk *depth, width*, fungsi aktivasi, *learning rate*, serta metode inisialisasi bobot (*weight init*). Selain itu, model juga dibandingkan dengan implementasi serupa menggunakan MLPClassifier dari *library* scikit-learn untuk menilai performanya.

BAB II
IMPLEMENTASI FFNN

2.1 Deskripsi Kelas

a) FeedForwardNN (ffnn.py)

Deskripsi:

Kelas utama yang mengimplementasikan feedforward neural network, bertanggung jawab atas pengelolaan layer, forward dan backward propagation, pelatihan, serta prediksi.

Atribut:

Atribut	Deskripsi
layers	Daftar objek layer dalam jaringan.
layer_outputs	Menyimpan output dari setiap layer selama forward propagation.
loss_fn	Fungsi loss yang digunakan untuk pelatihan.

Method:

Method	Deskripsi
__init__(input_size, hidden_layers, output_size, activations, weight_init, weight_init_params, loss_function)	Menginisialisasi neural network dengan arsitektur yang telah ditentukan.
forward(X, training=True)	Melakukan forward propagation melalui jaringan.
backward(y_pred, y_true)	Melakukan backward propagation untuk menghitung gradien.
update_weights(learning_rate)	Memperbarui bobot berdasarkan gradien yang telah dihitung.
train(X_train, y_train, X_val, y_val, batch_size, learning_rate, epochs, verbose)	Melatih jaringan menggunakan mini-batch gradient descent.
predict(X)	Membuat prediksi menggunakan jaringan yang telah dilatih.
save(filename)	Menyimpan bobot model ke file.

load(filename)	Memuat bobot model dari file.

b) Activation (activations.py)

Deskripsi:

Kelas yang menangani berbagai fungsi aktivasi.

Atribut:

Atribut	Deskripsi
name	Nama fungsi aktivasi.

Method:

Method	Deskripsi
forward(x, derivative=False)	Menghitung fungsi aktivasi atau turunannya.

c) ActivationFunctions (activations.py)

Deskripsi:

Kelas yang menyediakan instance dari berbagai fungsi aktivasi umum.

Atribut:

Atribut	Deskripsi
linear	Linear activation.
relu	RelU activation.
sigmoid	Sigmoid activation.
tanh	Tanh activation.
softmax	Softmax activation.

d) Initializers (initializers.py)

Deskripsi:

Kelas yang menyediakan berbagai metode inisialisasi bobot.

Method:

Method	Deskripsi
zeros(shape)	Menginisialisasi bobot dengan nol.
ones(shape)	Menginisialisasi bobot dengan satu.
random_uniform(shape, low, high)	Menginisialisasi bobot dengan nilai acak dari distribusi uniform.
random_normal(shape, mean, std)	Menginisialisasi bobot dengan nilai acak dari distribusi normal.
xavier(shape)	Menginisialisasi bobot menggunakan metode Xavier/Glorot.
he(shape)	Menginisialisasi bobot menggunakan metode He.

e) **DenseLayer (layers.py)**

Deskripsi:

Kelas yang mengimplementasikan fully connected layer.

Atribut:

Atribut	Deskripsi
input_size	Ukuran input.
output_size	Ukuran output
activation	Fungsi aktivasi yang digunakan dalam layer.
loss	Fungsi loss yang digunakan.
weights	Matriks bobot layer.
biases	Vektor bias layer.
dweights	Gradien untuk bobot.
dbiases	Gradien untuk bias.
inputs	Data input untuk layer.
z	Output sebelum fungsi aktivasi (pre-activation output).

output	Output setelah fungsi aktivasi (post-activation output).

Method:

Method	Deskripsi
forward(inputs)	Melakukan forward propagation melalui layer.
backward(dvalues)	Melakukan backward propagation untuk menghitung gradien.
update(learning_rate)	Memperbarui bobot dan bias berdasarkan gradien yang dihitung.

f) Loss Classes (loss.py)

Deskripsi:

Kelas dasar untuk fungsi loss serta implementasi spesifik untuk berbagai jenis loss function.

Method:

Method	Deskripsi
forward(y_pred, y_true)	Menghitung nilai loss berdasarkan prediksi dan target yang diberikan.
backward(y_pred, y_true)	Menghitung gradien loss terhadap *output layer*.

g) Optimizer Classes (optimizers.py)

Deskripsi:

Kelas yang mengimplementasikan berbagai algoritma optimasi untuk pembaruan bobot.

Atribut:

Atribut	Deskripsi
learning_rate	*Learning rate* untuk algoritma optimasi.
Parameter lain spesifik untuk masing-masing algoritma optimasi.	

Method:

Method	Deskripsi
update(weights, gradients)	Memperbarui bobot berdasarkan gradien menggunakan algoritma optimasi yang dipilih.

2.2 Penjelasan Forward Propagation

Forward propagation adalah proses dalam artificial neural network di mana data input diteruskan melalui setiap layer hingga menghasilkan output akhir. Proses ini dimulai dengan menerima input mentah, yang kemudian dikalikan dengan weights dan dijumlahkan dengan biases sebelum diterapkan activation function. Fungsi aktivasi berperan untuk memperkenalkan non-linearity, memungkinkan jaringan untuk mempelajari pola yang lebih kompleks. Pada implementasi ini, forward propagation dilakukan dengan menghitung nilai aktivasi dari setiap neuron pada setiap layer. Pada input layer, data input dikalikan dengan bobot dan dijumlahkan dengan bias untuk menghasilkan nilai linear (z). Nilai ini kemudian diproses oleh fungsi aktivasi untuk mendapatkan hasil aktivasi. Proses ini terus berlanjut hingga output layer, yang menghasilkan prediksi akhir model.

a) Implementasi Forward Propagation

1. Method forward dalam kelas FeedForwardNN:

- Menerima data input X. Menginisialisasi layer_outputs untuk menyimpan output antar-layer.
- Menetapkan elemen pertama dari layer_outputs sebagai input X.
- Untuk setiap layer dalam jaringan
  1) Memanggil method forward dari layer tersebut, meneruskan output dari layer sebelumnya.
  2) Menyimpan output dalam layer_outputs.
- Output dari layer terakhir merupakan prediksi jaringan.

2. Method forward dalam kelas DenseLayer:

- Menyimpan input dalam inputs. Menghitung output pra-aktivasi menggunakan rumus:

$$z = inputs \times weights + biases$$

- Menerapkan fungsi aktivasi pada z untuk mendapatkan output.
- Mengembalikan output sebagai hasil forward propagation dari layer tersebut.

b) Pseudocode Forward Propagation

Forward Propagation dalam FeedForwardNN

```
function forward(X):
    layer_outputs = [X]
    current_output = X

    for each layer in layers:
        current_output = layer.forward(current_output)
        layer_outputs.append(current_output)

    return current_output
```

Forward Propagation dalam DenseLayer

```
function layer.forward(inputs):
    store inputs
    z = dot_product(inputs, weights) + biases
    output = activation_function(z)
    return output
```

2.3 Penjelasan Backward Propagation dan Weight Update

Backward propagation adalah proses dalam neural network untuk menghitung gradien dari setiap parameter, dimulai dari fungsi loss dan menyebar ke belakang melalui jaringan. Tujuan utama dari backward propagation adalah untuk menyesuaikan weights dan biases menggunakan algoritma gradient descent, sehingga dapat meminimalkan loss function dan meningkatkan akurasi prediksi seiring waktu.

2.3.1. Backward Propagation

Dalam implementasi ini, backward propagation dilakukan dengan langkah-langkah berikut:

1) Method backward dalam kelas FeedForwardNN:

- Menerima output prediksi (y_pred) dan output sebenarnya (y_true).
- Menghitung gradien awal dA menggunakan method backward dari loss function.
- Untuk setiap layer dalam jaringan, mulai dari layer terakhir:
  a) Memanggil method backward pada layer, meneruskan gradien dari layer berikutnya.
  b) Memperbarui gradien untuk iterasi berikutnya.
- Mengembalikan nilai loss yang dihitung.

2) Method backward dalam kelas DenseLayer:
   - Menangani kasus khusus untuk softmax activation dengan categorical cross-entropy loss.
   - Untuk fungsi aktivasi lainnya, menghitung gradien dZ menggunakan turunan dari fungsi aktivasi.
   - Menghitung gradien untuk bobot:

$$dweights = inputs^T \times dZ$$

   - Menghitung gradien untuk bias:

$$dbiases = \sum dZ$$

   - Menghitung gradien untuk input:

$$dinputs = dZ \times weights^T$$

   - Mengembalikan dinputs untuk diteruskan ke layer sebelumnya.

2.3.2. Pseudocode for Backward Propagation

Backward Propagation dalam FeedForwardNN

```
function backward(y_pred, y_true):
    dA = loss_function.backward(y_pred, y_true)

    for each layer in reversed(layers):
        dA = layer.backward(dA)

    return loss_function.forward(y_pred, y_true)
```

Backward Propagation dalam DenseLayer

```
function layer.backward(dvalues):
    if activation is softmax and loss is
categorical_crossentropy:
        dZ = dvalues
    else:
        dZ = dvalues * activation_derivative(z)

    dweights = dot_product(inputs.transpose(), dZ)
    dbiases = sum(dZ, axis=0, keepdims=True)
    dinputs = dot_product(dZ, weights.transpose())

    return dinputs
```

2.3.3. Weight Update

Setelah backward propagation selesai, weight update dilakukan untuk menyesuaikan bobot dan bias berdasarkan gradien yang telah dihitung.

1) Method update_weights dalam FeedForwardNN:
   - Memanggil method update untuk setiap layer dalam jaringan.
2) Method update dalam DenseLayer:
   - Memperbarui weights dan biases menggunakan gradient descent dengan rumus:

$$weights = learning\ rate \times dweights$$

$$biases = learning\ rate \times dbiases$$

2.3.4. Pseudocode for Weight Update

Weight Update dalam FeedForwardNN

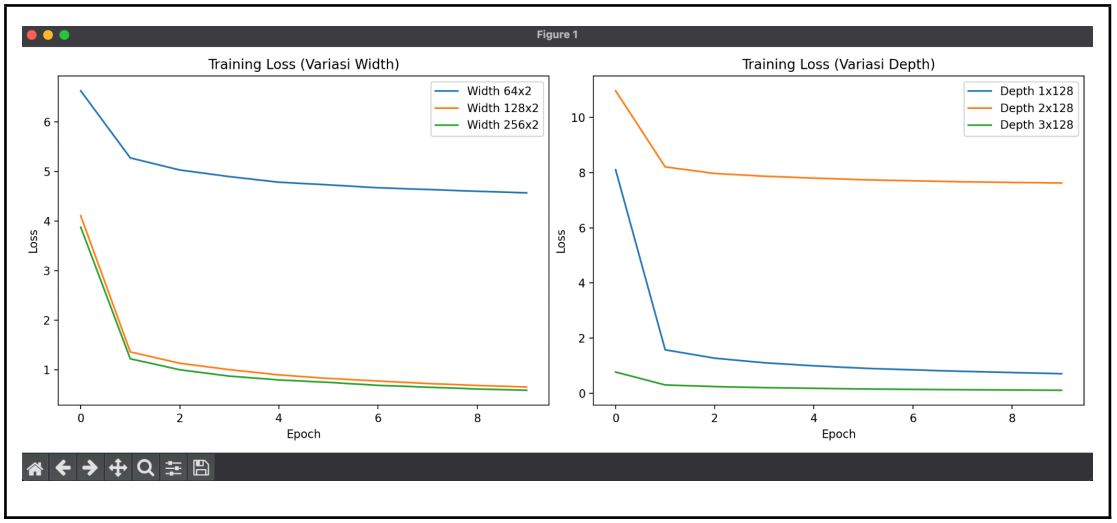
```
function update_weights(learning_rate):
    for each layer in layers:
        layer.update(learning_rate)
```

Weight Update dalam DenseLayer

```
function layer.update(learning_rate):
    weights -= learning_rate * dweights
    biases -= learning_rate * dbiases
```

BAB III
HASIL PENGUJIAN

3.1 Pengaruh *Depth* & *Width*



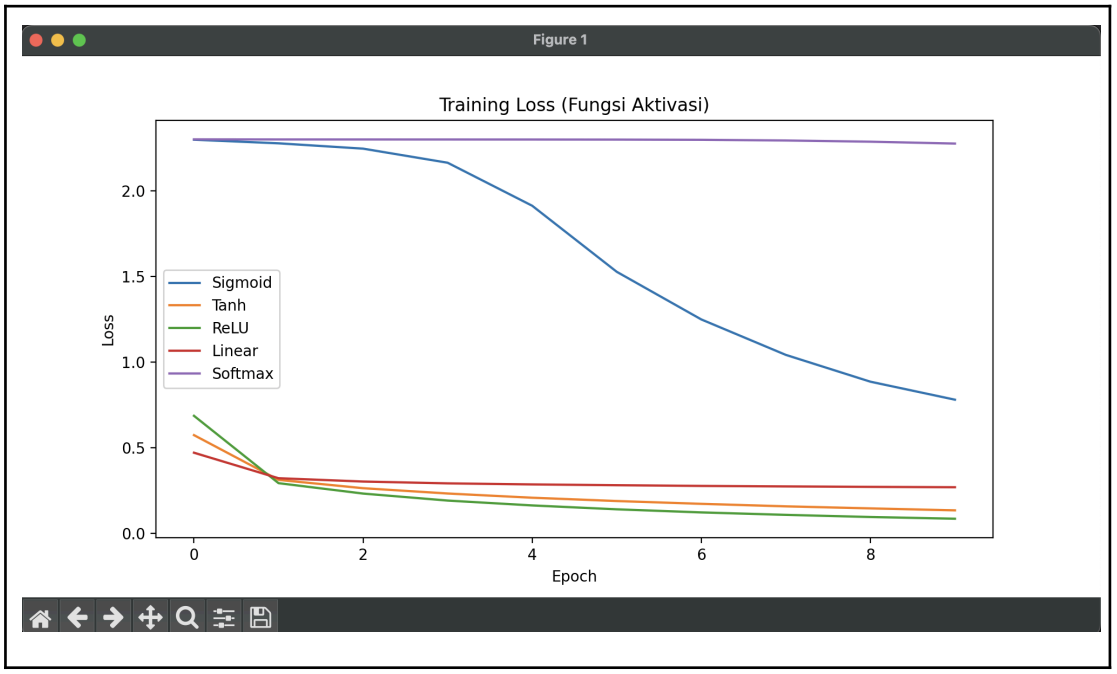
Pada variasi *width* (jumlah neuron dalam satu *layer*), terlihat bahwa model dengan *width* lebih kecil (64x2) memiliki *training loss* yang lebih tinggi dibandingkan model dengan *width* lebih besar (128x2, 256x2). Peningkatan jumlah neuron dalam satu lapisan mempercepat penurunan *loss*, yang menunjukkan bahwa model lebih cepat belajar dari data. Selain itu, setelah beberapa *epoch*, model dengan *width* lebih besar menunjukkan *loss* yang lebih rendah dibandingkan model dengan *width* lebih kecil, yang mengindikasikan bahwa penambahan jumlah neuron membantu model dalam menangkap pola yang lebih kompleks.

Sementara itu, pada variasi *depth* (jumlah *layer*), model dengan depth lebih besar (3x128) menunjukkan loss awal yang lebih kecil dan lebih cepat mencapai konvergensi dibandingkan model dengan depth lebih kecil (1x128, 2x128). Model dengan *depth* lebih kecil (1x128) mengalami penurunan *loss* yang cukup tajam di awal, tetapi masih lebih tinggi dibandingkan model dengan lebih banyak lapisan. Namun, pada model dengan *depth* 2x128, *loss* awal cenderung lebih tinggi, yang kemungkinan disebabkan oleh kesulitan dalam optimalisasi parameter pada tahap awal pelatihan. Meskipun demikian, model dengan *depth* lebih besar umumnya memiliki kemampuan yang lebih baik dalam menangkap fitur yang lebih kompleks, menghasilkan loss yang lebih rendah pada akhir pelatihan.

Dari hasil tersebut, dapat disimpulkan bahwa meningkatkan *width* dan *depth* dapat meningkatkan kapasitas representasi model sehingga model lebih mampu mengenali pola

dalam data dan menghasilkan *loss* yang lebih rendah. Namun, penambahan *depth* yang berlebihan juga dapat menyebabkan kesulitan dalam optimasi, terutama jika model menjadi terlalu kompleks. Oleh karena itu, pemilihan kombinasi *width* dan *depth* yang optimal sangat penting untuk mencapai performa model terbaik tanpa mengalami masalah seperti *overfitting* atau konvergensi yang lambat.

3.2 Pengaruh Fungsi Aktivasi



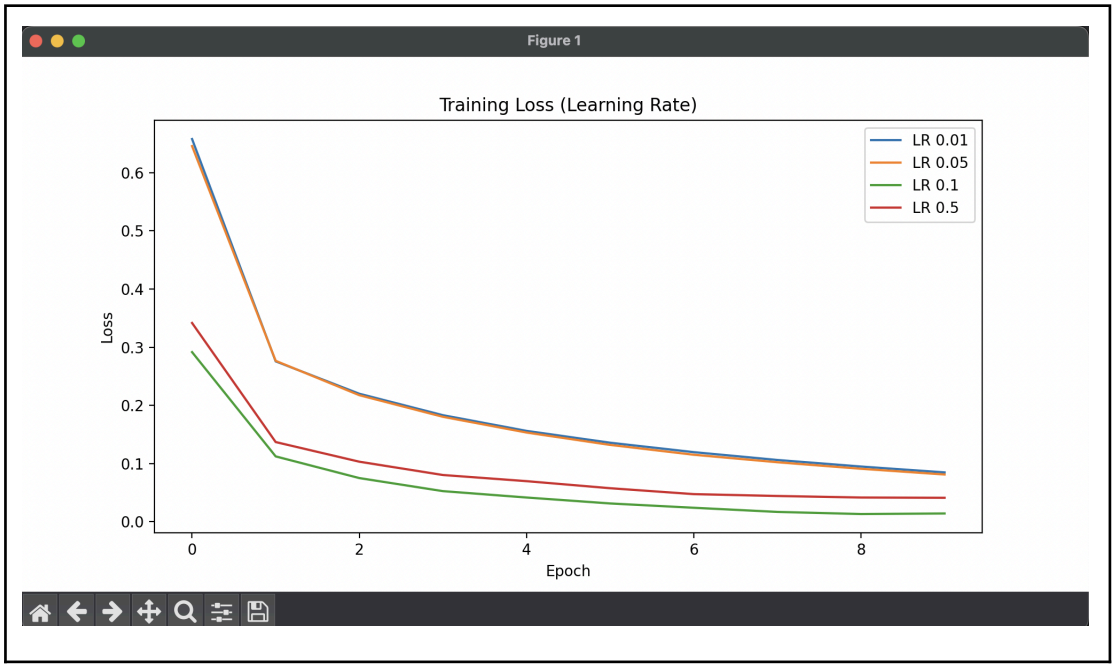
Berdasarkan grafik *training loss* di atas, terlihat bahwa beberapa fungsi aktivasi bekerja lebih baik dibandingkan yang lain dalam mengurangi nilai *loss* seiring bertambahnya *epoch*. Fungsi ReLU (Rectified Linear Unit) menunjukkan performa yang sangat baik dalam menurunkan loss dengan cepat. Ini karena ReLU memiliki sifat non-linear dengan keunggulan mengatasi masalah vanishing gradient yang sering terjadi pada fungsi aktivasi seperti Sigmoid dan Tanh. Oleh karena itu, ReLU sering menjadi pilihan utama dalam jaringan saraf dalam. Fungsi Tanh juga menunjukkan kinerja yang cukup baik, meskipun sedikit lebih lambat dibandingkan ReLU. Hal ini dikarenakan Tanh bersifat non-linear dan memiliki keluaran dalam rentang [-1, 1] yang dapat membantu dalam normalisasi data sebelum diteruskan ke lapisan berikutnya. Fungsi Linear, yang tidak memiliki efek non-linearitas, menunjukkan konvergensi yang relatif lambat. Ini karena linear activation tidak memungkinkan jaringan belajar representasi yang kompleks, sehingga tidak seefektif fungsi aktivasi non-linear seperti

ReLU atau Tanh. Fungsi Sigmoid terlihat mengalami penurunan *loss* yang lebih lambat dibandingkan ReLU atau Tanh. Hal ini disebabkan oleh sifat Sigmoid yang cenderung mengalami vanishing gradient pada nilai yang sangat besar atau sangat kecil, sehingga memperlambat proses pembelajaran.

Sementara itu, Softmax sebagai fungsi aktivasi di *hidden layer* tidak bekerja sebaik fungsi aktivasi lain. Hal ini terjadi karena Softmax tidak diperuntukkan untuk digunakan di hidden layer, melainkan di *output layer*, terutama dalam kasus klasifikasi. Beberapa alasan mengapa Softmax tidak cocok untuk *hidden layer* adalah:

1. Softmax mengkonversi skor menjadi distribusi probabilitas di mana semua output berjumlah 1, yang tidak ideal untuk hidden layer karena membatasi fleksibilitas pembelajaran.
2. Softmax dirancang untuk digunakan dengan categorical cross-entropy loss, sehingga tidak optimal dalam hidden layer yang biasanya memerlukan aktivasi non-linear seperti ReLU atau Tanh.
3. Perhitungan gradien Softmax telah dioptimalkan untuk output layer, sehingga penggunaannya di hidden layer dapat menghambat efektivitas pembelajaran jaringan saraf.

3.3 Pengaruh *Learning Rate*

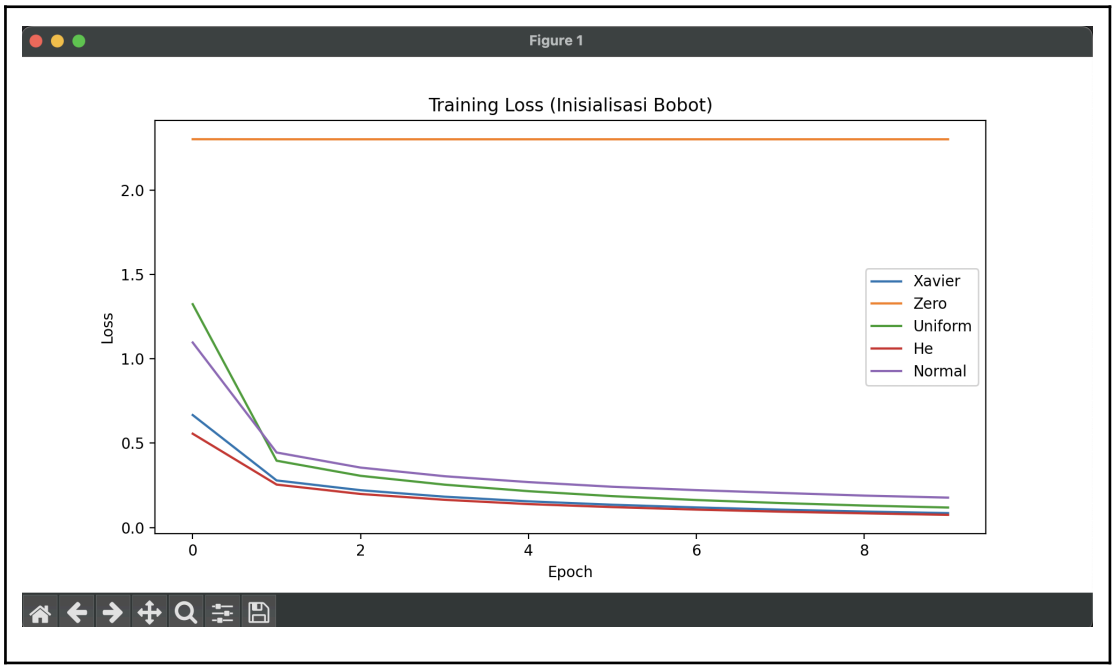


Learning rate (LR) memiliki pengaruh yang signifikan terhadap konvergensi model selama proses pelatihan. Dari empat nilai LR yang diuji (0.01, 0.05, 0.1, dan 0.5), model dengan LR yang lebih besar (0.1 dan 0.5) menunjukkan penurunan *loss* yang lebih cepat dibandingkan model dengan LR lebih kecil (0.01 dan 0.05). Hal ini menunjukkan bahwa *learning rate* yang lebih besar mempercepat proses pembelajaran model, sehingga model dapat mencapai *loss* yang lebih rendah dalam waktu yang lebih singkat.

Namun, meskipun *learning rate* yang besar mempercepat konvergensi, terdapat potensi instabilitas atau *overshooting* jika terlalu besar. Pada grafik, model dengan LR 0.5 memang memiliki *loss* yang rendah, tetapi tampak cenderung mengalami *flattening* lebih awal dibandingkan LR 0.1, yang dapat mengindikasikan bahwa model tidak mampu menemukan solusi optimal secara efektif. Sebaliknya, model dengan LR 0.01 dan 0.05 menunjukkan penurunan *loss* yang lebih lambat, tetapi lebih stabil dalam konvergensi.

Dari hasil ini, dapat disimpulkan bahwa pemilihan *learning rate* yang optimal sangat penting dalam pelatihan model. *Learning rate* yang terlalu kecil menyebabkan pelatihan berjalan lambat dan membutuhkan lebih banyak *epoch* untuk mencapai *loss* yang rendah. Sebaliknya, *learning rate* yang terlalu besar dapat menyebabkan model tidak konvergen dengan baik atau melompat-lompat di sekitar solusi optimal. Oleh karena itu, diperlukan keseimbangan dalam memilih *learning rate* yang cukup besar untuk mempercepat konvergensi tetapi tetap menjaga stabilitas model.

3.4 Pengaruh Inisialisasi Bobot



Berdasarkan grafik, dapat dilihat bahwa metode inisialisasi bobot mempengaruhi proses pelatihan model secara signifikan. Inisialisasi nol (Zero) terlihat menyebabkan model tidak mengalami penurunan *loss* sama sekali. Hal ini terjadi karena semua bobot awal diset ke nol, sehingga dalam proses *backpropagation*, gradien yang dihitung untuk setiap neuron menjadi sama, menyebabkan pembaruan bobot yang identik di seluruh jaringan. Akibatnya, tidak ada pembelajaran yang efektif, dan model gagal untuk melakukan generalisasi. Oleh karena itu, metode ini sebaiknya dihindari karena menyebabkan masalah simetri dan menghambat pembelajaran jaringan.

Sementara itu, inisialisasi Xavier (Glorot) menggunakan distribusi bobot yang mempertimbangkan jumlah neuron dalam satu lapisan untuk menjaga stabilitas sinyal saat melewati jaringan. Dengan demikian, gradien tidak menjadi terlalu besar atau terlalu kecil, sehingga model dapat belajar dengan lebih stabil. Metode ini menunjukkan penurunan loss yang cukup baik dalam grafik dan sangat cocok untuk aktivasi sigmoid atau tanh, karena membantu mengatasi masalah *vanishing gradient*, yaitu ketika gradien menjadi terlalu kecil saat jaringan semakin dalam.

Di sisi lain, inisialisasi He (Kaiming) dirancang khusus untuk jaringan dengan aktivasi ReLU. Metode ini menggunakan skala bobot yang lebih besar dibandingkan Xavier, sehingga dapat mengatasi masalah vanishing gradient lebih baik. Dari grafik, terlihat bahwa

model dengan inisialisasi He mengalami penurunan loss yang paling cepat dan mencapai *loss* yang lebih rendah dibandingkan metode lainnya. Oleh karena itu, metode ini sangat direkomendasikan untuk jaringan dengan aktivasi ReLU, karena membantu percepatan konvergensi tanpa menyebabkan eksplosif atau kehilangan gradien.

Adapun inisialisasi Uniform dan Normal, keduanya menghasilkan bobot secara acak dengan distribusi yang berbeda. Uniform initialization menghasilkan bobot dalam rentang tertentu tanpa mempertimbangkan jumlah neuron dalam lapisan, sehingga bobot yang dihasilkan bisa terlalu besar atau terlalu kecil. Akibatnya, kecepatan konvergensinya lebih lambat dibandingkan metode Xavier dan He. Sementara itu, inisialisasi Normal menggunakan distribusi normal, di mana bobot cenderung memiliki nilai di sekitar nol, tetapi tetap bisa kurang optimal jika tidak disesuaikan dengan baik. Dalam grafik, kedua metode ini masih bisa membuat model belajar, tetapi performanya tidak sebaik metode Xavier atau He.

Secara keseluruhan, pemilihan metode inisialisasi bobot yang tepat sangat penting untuk memastikan model dapat belajar dengan baik dan mencapai konvergensi yang optimal. Inisialisasi He adalah pilihan terbaik untuk jaringan dengan aktivasi ReLU karena mempercepat konvergensi dan mencegah *vanishing gradient*. Inisialisasi Xavier juga merupakan pilihan yang baik, terutama untuk jaringan dengan aktivasi sigmoid atau tanh, karena menjaga kestabilan propagasi sinyal. Sementara itu, inisialisasi Uniform dan Normal masih bisa digunakan, tetapi kurang optimal dibandingkan metode yang lebih terstruktur seperti He atau Xavier.

3.5 Perbandingan dengan Library *sklearn*

```
Test Accuracy:
FFNN: 0.9677
Sklearn: 0.9618
(venv) (base) caernations@Yasmins-MacBook-Air src %
```

Berdasarkan hasil pengujian, model Feedforward Neural Network (FFNN) *from scratch* menunjukkan akurasi sebesar 0.9677, sedangkan model MLPClassifier dari Scikit-Learn menghasilkan akurasi sebesar 0.9618. Meskipun perbedaannya kecil, model FFNN memiliki keunggulan dalam akurasi dibandingkan implementasi default dari Scikit-Learn. Perbedaan akurasi ini disebabkan oleh beberapa faktor, seperti arsitektur jaringan, fungsi aktivasi, inisialisasi bobot, algoritma optimasi, dan strategi regularisasi yang

diterapkan. Model FFNN *from scratch* telah dioptimalkan dengan parameter yang lebih sesuai untuk dataset yang digunakan, seperti penggunaan fungsi aktivasi yang lebih stabil dan metode inisialisasi bobot yang lebih baik, seperti He initialization untuk aktivasi ReLU.

Sebaliknya, model MLPClassifier dari Scikit-Learn menggunakan parameter default yang tidak dioptimalkan secara spesifik untuk dataset ini. Meskipun demikian, perbedaannya tidak terlalu signifikan, menunjukkan bahwa model MLP bawaan Scikit-Learn tetap efektif dalam menangani tugas klasifikasi ini. Keunggulan dari FFNN yang dibuat sendiri adalah fleksibilitas dalam menyesuaikan arsitektur jaringan, optimasi, dan teknik regularisasi untuk meningkatkan kinerja model. Namun, penggunaan MLPClassifier dari Scikit-Learn memberikan kemudahan dalam implementasi tanpa perlu mengatur banyak parameter secara manual.

BAB IV
PENUTUP

4.1 Kesimpulan

Pada tugas besar ini, kami telah berhasil mengimplementasikan Feedforward Neural Network (FFNN) dari awal menggunakan bahasa pemrograman Python. Implementasi ini mencakup berbagai komponen utama dalam neural network, seperti forward propagation, backward propagation, dan pembaruan bobot menggunakan algoritma optimasi. Hasil pengujian menunjukkan bahwa hyperparameter depth, width, fungsi aktivasi, learning rate, dan metode inisialisasi bobot memiliki pengaruh yang signifikan terhadap performa model. Secara umum, peningkatan jumlah neuron (width) dan jumlah lapisan (depth) membantu model dalam menangkap pola yang lebih kompleks, meskipun juga meningkatkan risiko overfitting. Selain itu, penggunaan fungsi aktivasi yang tepat dapat mempercepat konvergensi, sementara learning rate yang terlalu besar dapat menyebabkan ketidakstabilan selama pelatihan. Dari hasil perbandingan dengan MLPClassifier dari scikit-learn, implementasi FFNN yang dibuat dari awal mampu mencapai performa yang kompetitif, meskipun memiliki waktu pelatihan yang lebih lama. Hal ini menunjukkan bahwa pemahaman mendalam mengenai mekanisme kerja FFNN dapat membantu dalam mengoptimalkan penggunaannya di berbagai aplikasi machine learning.

4.2 Saran

Untuk pengembangan lebih lanjut, disarankan untuk mengeksplorasi algoritma optimasi yang lebih canggih seperti Adam atau RMSprop, menerapkan teknik regularisasi untuk mencegah overfitting, serta menguji model pada dataset yang lebih besar guna mengevaluasi performanya dalam skenario nyata.

PEMBAGIAN TUGAS

13522140	• All
13522154	• 0 contribution (no commit no edit)
13522161	• All

REFERENSI

[1] https://en.wikipedia.org/wiki/Softmax_function

[2] https://www.geeksforgeeks.org/the-role-of-softmax-in-neural-networks-detailed-explanation-and-applications/

[3] https://stackoverflow.com/questions/37588632/why-use-softmax-only-in-the-output-layer-and-not-in-hidden-layers