Ahmet Bera Özbolat
21902777

EEE 443-Neural Networks

Mini Project 3-HAR

INTRODUCTION

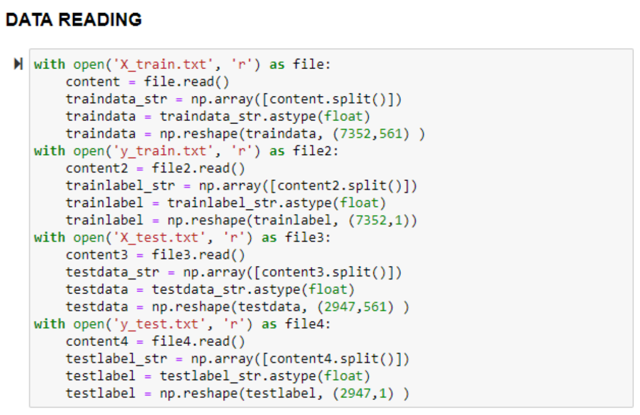
In this project, our primary objective is to develop a robust neural network model tailored for the recognition of six distinct human activities. We utilize data acquired from a Samsung Galaxy S2 smartphone's gyroscope and accelerometer sensors, while also accounting for gravity's impact by generating a third dataset. From these three data sources, we extract crucial statistical features including mean, standard deviation, variance, skewness, and kurtosis, amounting to a total of 561 input data points for each activity pattern. Our dataset comprises 7,352 training samples and 2,947 testing samples.

The neural network architecture we employ consists of one input layer, followed by two hidden layers, and culminating in an output layer. The first hidden layer encompasses 300 neurons, while the second hidden layer varies between 100 and 200 neurons in different scenarios. We explore the effects of two distinct learning coefficients (0.01 and 0.001), two momentum values (0 and 0.9), and two mini-batch sizes (0 and 50), resulting in a total of 16 different configurations. Additionally, we introduce a 17th case, optimizing for the best performance by incorporating a dropout rate of p = 0.5.

This project offers an innovative approach to activity recognition by leveraging key statistical features from sensor data, mitigating the challenges associated with recurrent neural networks such as vanishing and exploding gradients. Our neural network architecture, driven by ReLU activation in hidden layers and softmax activation in the output layer, is evaluated across these various configurations, allowing us to assess training error, test accuracies, and identify misclassified patterns. The culmination of this research will unveil the most accurate configuration, which we will further enhance through dropout regularization, thereby bolstering the model's predictive capabilities.

IMPLEMENTATION

The first step involves accessing the data for both training and testing. To achieve this, the following code has been developed for extracting data from files.

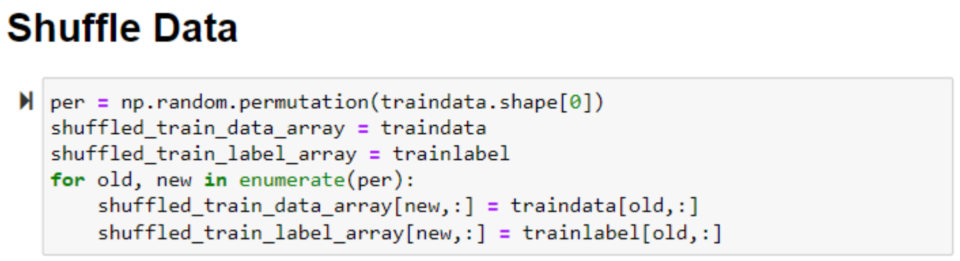


DATA READING

```
with open('X_train.txt', 'r') as file:
    content = file.read()
    traindata_str = np.array([content.split()])
    traindata = traindata_str.astype(float)
    traindata = np.reshape(traindata, (7352,561) )
with open('y_train.txt', 'r') as file2:
    content2 = file2.read()
    trainlabel_str = np.array([content2.split()])
    trainlabel = trainlabel_str.astype(float)
    trainlabel = np.reshape(trainlabel, (7352,1))
with open('X_test.txt', 'r') as file3:
    content3 = file3.read()
    testdata_str = np.array([content3.split()])
    testdata = testdata_str.astype(float)
    testdata = np.reshape(testdata, (2947,561) )
with open('y_test.txt', 'r') as file4:
    content4 = file4.read()
    testlabel_str = np.array([content4.split()])
    testlabel = testlabel_str.astype(float)
    testlabel = np.reshape(testlabel, (2947,1) )
```

Figure 1:Data Reading Function

Additionally, shuffling the data plays a crucial role in preventing potential errors that may occur during the training process, as we've learned from previous projects.

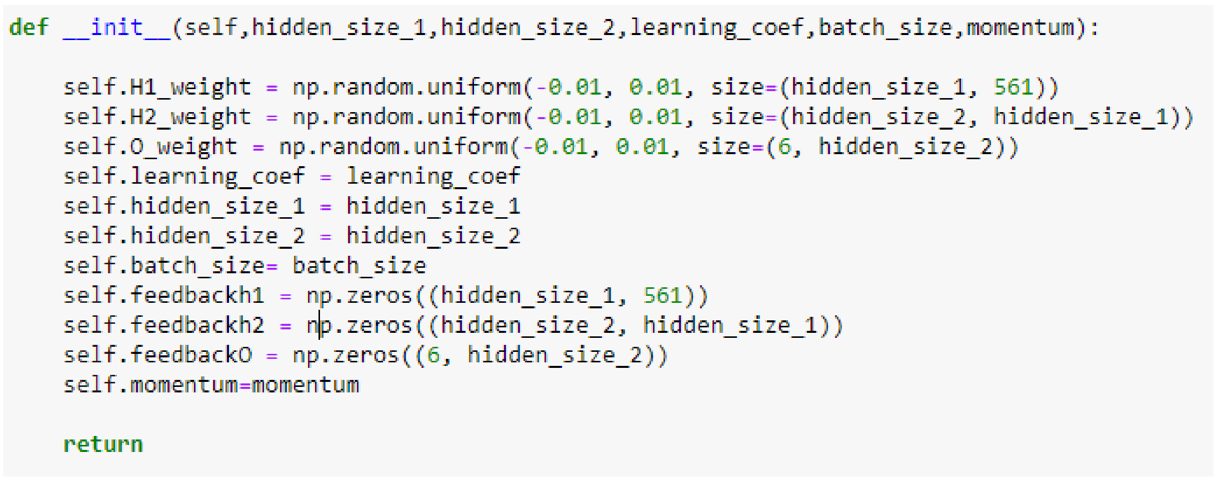


Shuffle Data

```
per = np.random.permutation(traindata.shape[0])
shuffled_train_data_array = traindata
shuffled_train_label_array = trainlabel
for old, new in enumerate(per):
    shuffled_train_data_array[new,:] = traindata[old,:]
    shuffled_train_label_array[new,:] = trainlabel[old,:]
```

Figure 2: Shuffle Data Code

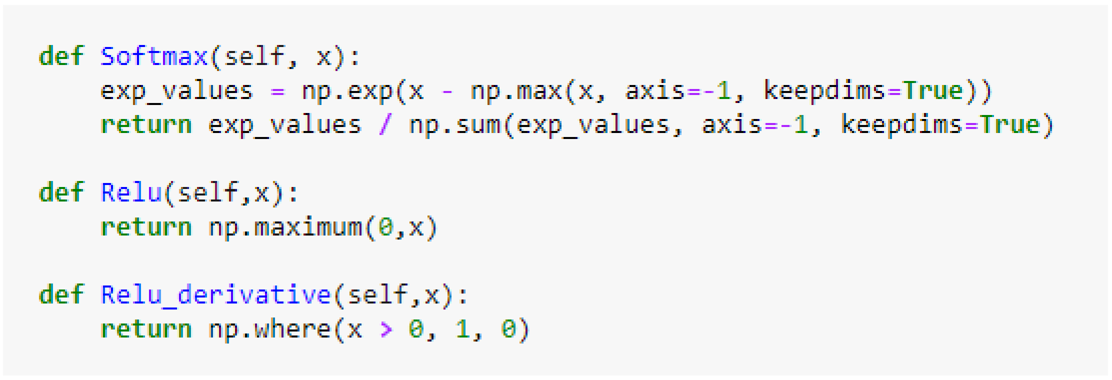
I began by creating a class in Python. The initial function within this class is responsible for randomly initializing network weights within the range of (-0.01, 0.01). This function also stores critical information such as neuron counts in the hidden layers, the learning coefficient, mini-batch size, and momentum specific to the particular case in the class.



```
def __init__(self,hidden_size_1,hidden_size_2,learning_coef,batch_size,momentum):

    self.H1_weight = np.random.uniform(-0.01, 0.01, size=(hidden_size_1, 561))
    self.H2_weight = np.random.uniform(-0.01, 0.01, size=(hidden_size_2, hidden_size_1))
    self.O_weight = np.random.uniform(-0.01, 0.01, size=(6, hidden_size_2))
    self.learning_coef = learning_coef
    self.hidden_size_1 = hidden_size_1
    self.hidden_size_2 = hidden_size_2
    self.batch_size= batch_size
    self.feedbackh1 = np.zeros((hidden_size_1, 561))
    self.feedbackh2 = np.zeros((hidden_size_2, hidden_size_1))
    self.feedbackO = np.zeros((6, hidden_size_2))
    self.momentum=momentum

    return
```

Figure 3:Weight Initilaze Function

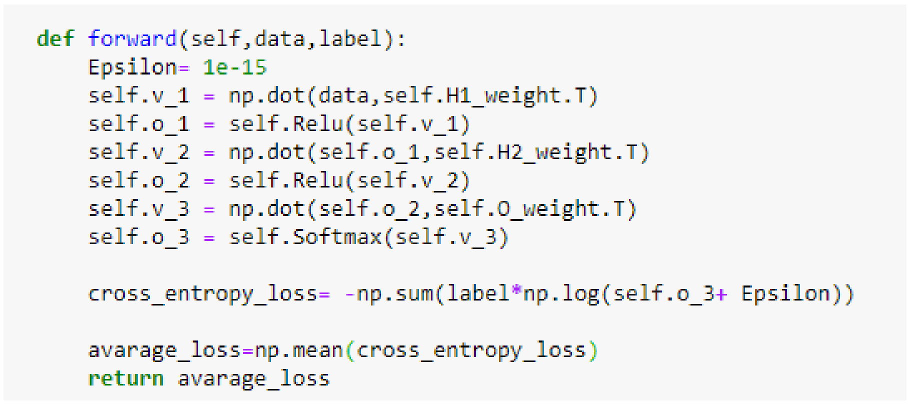
To create the neural network, the following activation functions were necessary. Their functionality was developed with the assistance of lecture notes.



```
def Softmax(self, x):
    exp_values = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return exp_values / np.sum(exp_values, axis=-1, keepdims=True)

def Relu(self,x):
    return np.maximum(0,x)

def Relu_derivative(self,x):
    return np.where(x > 0, 1, 0)
```

Figure 4: Activation Functions

This function represents the forward propagation step in my neural network project. It takes input data and corresponding labels, computes intermediate values and activations through hidden layers using the ReLU activation function, and finally, applies the Softmax activation function to produce output probabilities. The function calculates the cross-entropy loss between the predicted and actual labels, averages it, and returns the average loss as a measure of how well the neural network is performing during training



```python
def forward(self,data,label):
    Epsilon= 1e-15
    self.v_1 = np.dot(data,self.H1_weight.T)
    self.o_1 = self.Relu(self.v_1)
    self.v_2 = np.dot(self.o_1,self.H2_weight.T)
    self.o_2 = self.Relu(self.v_2)
    self.v_3 = np.dot(self.o_2,self.O_weight.T)
    self.o_3 = self.Softmax(self.v_3)

    cross_entropy_loss= -np.sum(label*np.log(self.o_3+ Epsilon))

    avarage_loss=np.mean(cross_entropy_loss)
    return avarage_loss
```

Figure 5: Forward Propagation Function

Given that we are working with categorical cross-entropy, the error is computed using this specific loss function, as outlined in the lecture notes.

$$\boxed{E_x = -\sum_{j=1}^{R} d_{kj} \log(y_k) \rightarrow -\log(y_k)}$$

Categorical Cross Entropy
Softmax Loss
Log-likelihood cost

$$\boxed{E = \frac{1}{N}\sum_x E_x}$$

Figure 6:Categorial Cross Entropy From Lecture Notes

Utilizing the Python 'feature' class, we store the outputs of each hidden layer and their corresponding vectors during each iteration. These stored data will be instrumental in calculating gradient descents for weight updates in the subsequent steps.

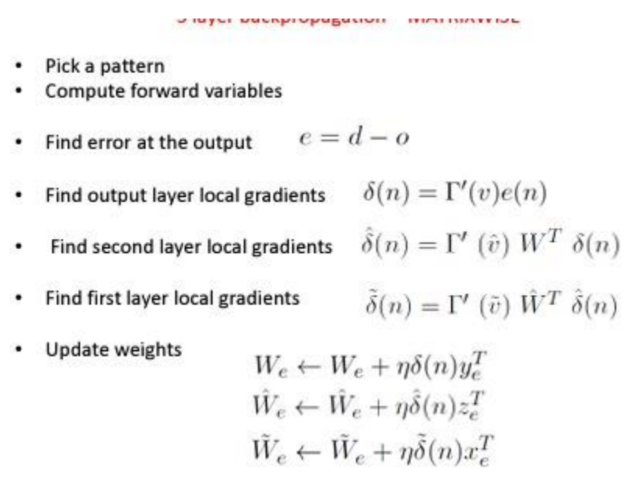
In the provided code, the gradient descent for the output layer is calculated as follows: “v3grad” represents the error in the output layer, obtained by subtracting the predicted values from the actual labels. Then, “outgrad” is computed by taking the dot product between the transposed “v3grad” and the activations from the second hidden layer. The gradient is computed according to the formula outlined in the lecture notes for both the one-hot encoded 1 output class and the 0 output classes.

$$\frac{\partial E}{\partial w_{1j}} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial v_1}\frac{\partial v_1}{\partial w_{1j}}$$
$$\frac{\partial E}{\partial y_i} = -\frac{1}{y_i}$$
$$\frac{\partial y_i}{\partial v_1} = -\frac{e^{v_1}e^{v_i}}{(\sum_j e^{v_y})^2} = -y_i y_1$$
$$\frac{\partial E}{\partial w_{1j}} = \frac{1}{y_i} y_i y_1 x_j = y_1 x_j$$
$$w_{1j} \leftarrow w_{1j} \leftarrow w_{1j} - \eta y_1 x_j$$

$$\frac{\partial E}{\partial w_{1j}} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial v_1}\frac{\partial v_1}{\partial w_{1j}}$$
$$\frac{\partial E}{\partial y_i} = -\frac{1}{y_i}$$
$$\frac{\partial y_i}{\partial v_1} = -\frac{e^{v_1}e^{v_i}}{(\sum_j e^{v_y})^2} = -y_i y_1$$
$$\frac{\partial E}{\partial w_{1j}} = \frac{1}{y_i} y_i y_1 x_j = y_1 x_j$$
$$w_{1j} \leftarrow w_{1j} \leftarrow w_{1j} - \eta y_1 x_j$$

Figure 7: Gradient Calculations for Cross-Entropy Softmax From Lecture Notes

The gradient of the second hidden layer is determined following a similar logic employed in the previous project. It leverages the gradient information from the output layer to calculate the second gradient, as prescribed in the lecture notes.



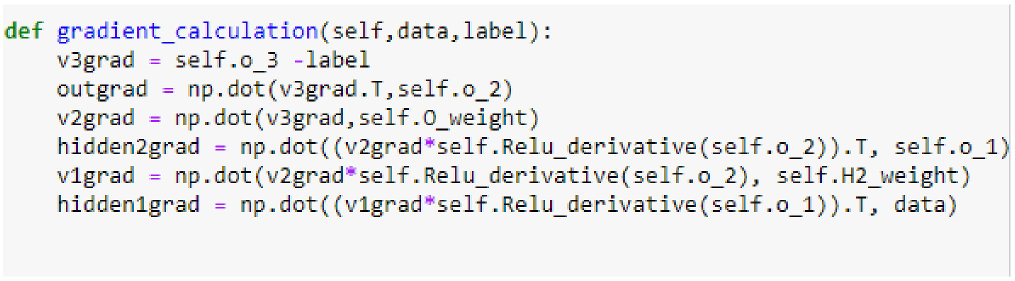
[illegible]

- Pick a pattern
- Compute forward variables
- Find error at the output $e = d - o$
- Find output layer local gradients $\delta(n) = \Gamma'(v)e(n)$
- Find second layer local gradients $\hat{\delta}(n) = \Gamma'(\hat{v})\, W^T\, \delta(n)$
- Find first layer local gradients $\tilde{\delta}(n) = \Gamma'(\tilde{v})\, \hat{W}^T\, \hat{\delta}(n)$
- Update weights $$W_e \leftarrow W_e + \eta\delta(n)y_e^T$$ $$\hat{W}_e \leftarrow \hat{W}_e + \eta\hat{\delta}(n)z_e^T$$ $$\tilde{W}_e \leftarrow \tilde{W}_e + \eta\tilde{\delta}(n)x_e^T$$

Figure 8: Back Propagation Algorithm From Lecture Notes

In the final step of the gradient descent calculation, the gradient for the first hidden layer is determined using the same logic as in the previous step. This continuity in approach ensures consistency and follows the established methodology for gradient calculation within the neural network.

The provided code represents the function responsible for calculating gradient descents, returning the gradients for each iteration.



```
def gradient_calculation(self,data,label):
    v3grad = self.o_3 -label
    outgrad = np.dot(v3grad.T,self.o_2)
    v2grad = np.dot(v3grad,self.O_weight)
    hidden2grad = np.dot((v2grad*self.Relu_derivative(self.o_2)).T, self.o_1)
    v1grad = np.dot(v2grad*self.Relu_derivative(self.o_2), self.H2_weight)
    hidden1grad = np.dot((v1grad*self.Relu_derivative(self.o_1)).T, data)
```

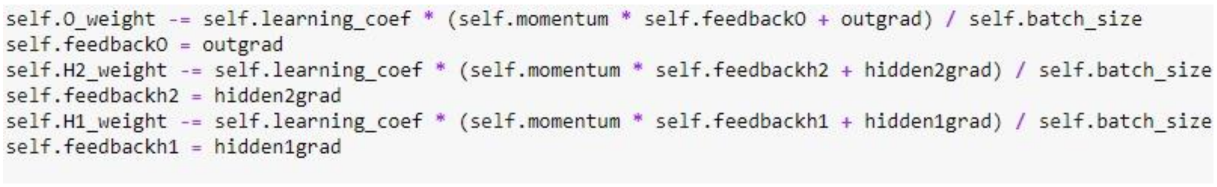
Figure 9: Gradient Calculation Function

For some cases in the project momentum will be used. Momentum in neural networks is a technique that improves training efficiency by introducing a momentum term, which is a running average of past gradients. It helps accelerate convergence and stabilizes training by allowing the network to move more swiftly toward the optimal solution when gradients consistently point in the same direction, while also dampening oscillations when gradients change direction.

$$\Delta W(n) = \underbrace{-\eta \frac{\partial E}{\partial W}}_{\text{Gradient Descent}} + \underbrace{\alpha \Delta W(n-1)}_{\text{Momentum term}}$$

Figure 10: Momentum method

The weight update process involves summing up the gradients of each batch. In online training, where the batch size is one (0 batches), weight updates are performed accordingly. Additionally, during weight updates, momentum is taken into account. It's important to note that in certain cases, momentum may be set to zero, depending on the specific configuration.

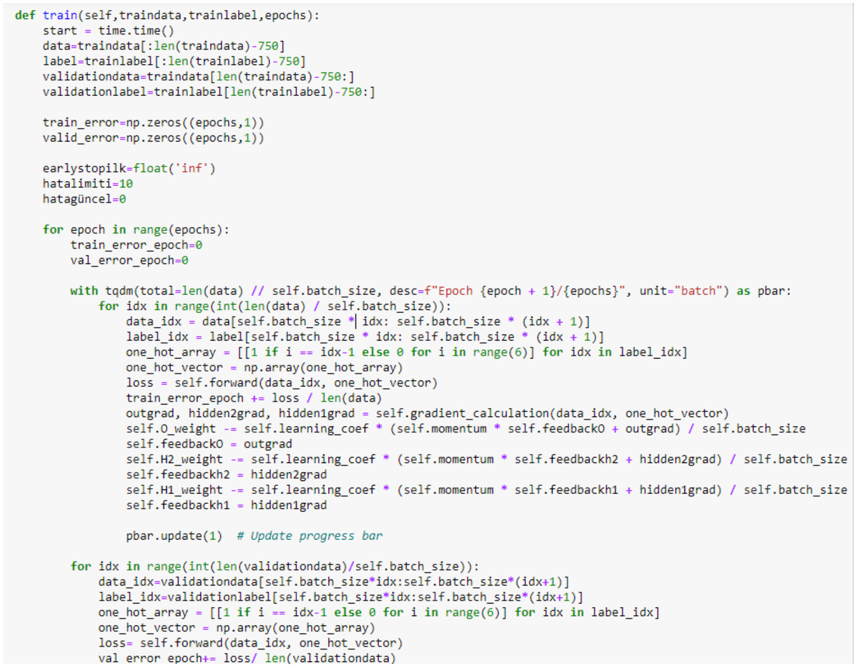


```
self.O_weight -= self.learning_coef * (self.momentum * self.feedbackO + outgrad) / self.batch_size
self.feedbackO = outgrad
self.H2_weight -= self.learning_coef * (self.momentum * self.feedbackh2 + hidden2grad) / self.batch_size
self.feedbackh2 = hidden2grad
self.H1_weight -= self.learning_coef * (self.momentum * self.feedbackh1 + hidden1grad) / self.batch_size
self.feedbackh1 = hidden1grad
```

Figure 11: Weight Update Function With Momentum

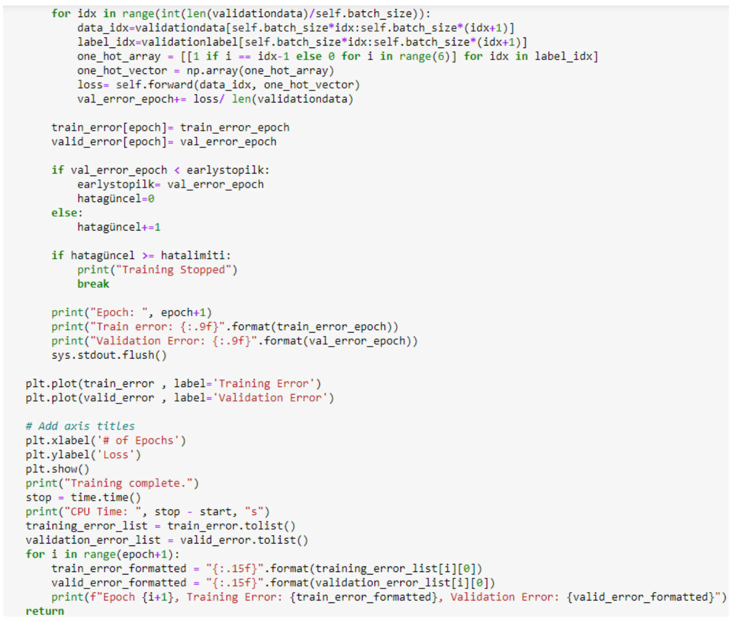
In the provided code above, the feedback term presents the previous gradient descent term.

During the training phase, before the data is used for training, 10% of the data is set aside for the validation set. In each training iteration, training errors and validation errors for each epoch are recorded and stored in a list. To implement early stopping, a loop iteration within the function checks if the validation error increases, essentially monitoring for a lack of improvement in training for a certain consecutive number of steps, initially determined. This mechanism ensures that the training algorithm halts when there is no observable progress. To enhance tracking, features like tqdm are incorporated to monitor progress, remaining time, and other relevant metrics. Upon completion of training, the validation error and training error are visualized through plotting.



```
def train(self,traindata,trainlabel,epochs):
    start = time.time()
    data=traindata[:len(traindata)-750]
    label=trainlabel[:len(trainlabel)-750]
    validationdata=traindata[len(traindata)-750:]
    validationlabel=trainlabel[len(trainlabel)-750:]

    train_error=np.zeros((epochs,1))
    valid_error=np.zeros((epochs,1))

    earlystopilk=float('inf')
    hatalimiti=10
    hatagüncel=0

    for epoch in range(epochs):
        train_error_epoch=0
        val_error_epoch=0

        with tqdm(total=len(data) // self.batch_size, desc=f"Epoch {epoch + 1}/{epochs}", unit="batch") as pbar:
            for idx in range(int(len(data) / self.batch_size)):
                data_idx = data[self.batch_size * idx: self.batch_size * (idx + 1)]
                label_idx = label[self.batch_size * idx: self.batch_size * (idx + 1)]
                one_hot_array = [[1 if i == idx-1 else 0 for i in range(6)] for idx in label_idx]
                one_hot_vector = np.array(one_hot_array)
                loss = self.forward(data_idx, one_hot_vector)
                train_error_epoch += loss / len(data)
                outgrad, hidden2grad, hidden1grad = self.gradient_calculation(data_idx, one_hot_vector)
                self.O_weight -= self.learning_coef * (self.momentum * self.feedbackO + outgrad) / self.batch_size
                self.feedbackO = outgrad
                self.H2_weight -= self.learning_coef * (self.momentum * self.feedbackh2 + hidden2grad) / self.batch_size
                self.feedbackh2 = hidden2grad
                self.H1_weight -= self.learning_coef * (self.momentum * self.feedbackh1 + hidden1grad) / self.batch_size
                self.feedbackh1 = hidden1grad

                pbar.update(1)  # Update progress bar

        for idx in range(int(len(validationdata)/self.batch_size)):
            data_idx=validationdata[self.batch_size*idx:self.batch_size*(idx+1)]
            label_idx=validationlabel[self.batch_size*idx:self.batch_size*(idx+1)]
            one_hot_array = [[1 if i == idx-1 else 0 for i in range(6)] for idx in label_idx]
            one_hot_vector = np.array(one_hot_array)
            loss= self.forward(data_idx, one_hot_vector)
            val_error_epoch+= loss/ len(validationdata)
```

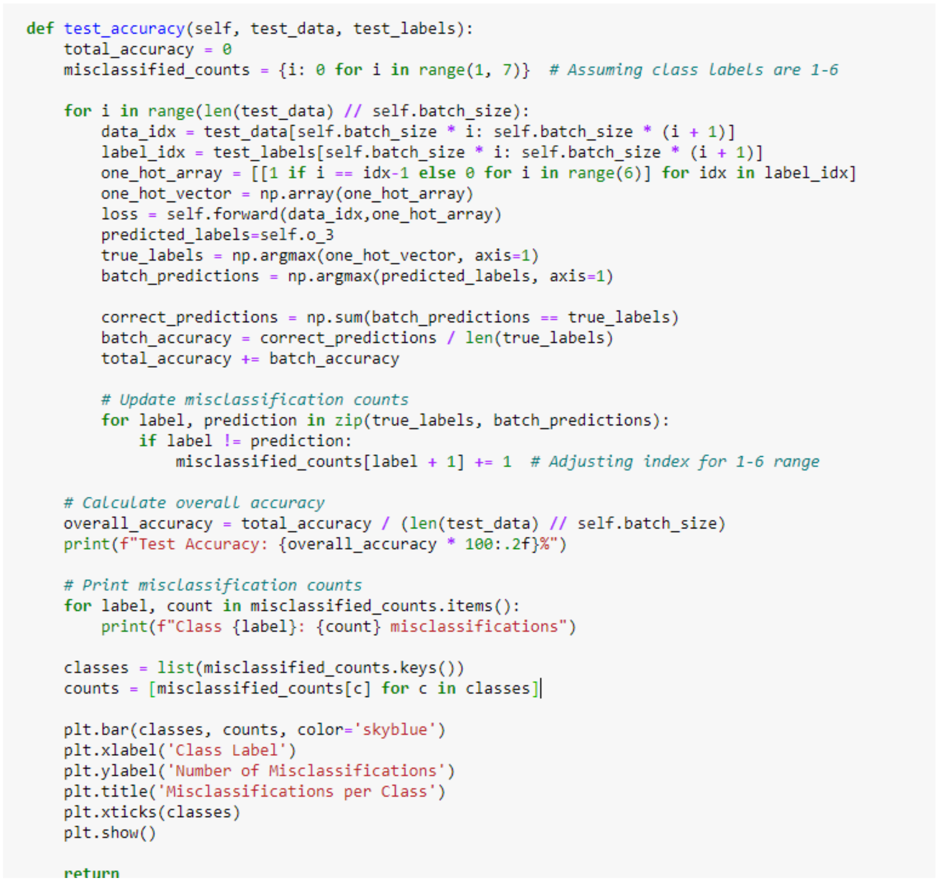
Figure 12: Training Function Part 1



```
        for idx in range(int(len(validationdata)/self.batch_size)):
            data_idx=validationdata[self.batch_size*idx:self.batch_size*(idx+1)]
            label_idx=validationlabel[self.batch_size*idx:self.batch_size*(idx+1)]
            one_hot_array = [[1 if i == idx-1 else 0 for i in range(6)] for idx in label_idx]
            one_hot_vector = np.array(one_hot_array)
            loss= self.forward(data_idx, one_hot_vector)
            val_error_epoch+= loss/ len(validationdata)

        train_error[epoch]= train_error_epoch
        valid_error[epoch]= val_error_epoch

        if val_error_epoch < earlystopilk:
            earlystopilk= val_error_epoch
            hatagüncel=0
        else:
            hatagüncel+=1

        if hatagüncel >= hatalimiti:
            print("Training Stopped")
            break

        print("Epoch: ", epoch+1)
        print("Train error: {:.9f}".format(train_error_epoch))
        print("Validation Error: {:.9f}".format(val_error_epoch))
        sys.stdout.flush()

    plt.plot(train_error , label='Training Error')
    plt.plot(valid_error , label='Validation Error')

    # Add axis titles
    plt.xlabel('# of Epochs')
    plt.ylabel('Loss')
    plt.show()
    print("Training complete.")
    stop = time.time()
    print("CPU Time: ", stop - start, "s")
    training_error_list = train_error.tolist()
    validation_error_list = valid_error.tolist()
    for i in range(epoch+1):
        train_error_formatted = "{:.15f}".format(training_error_list[i][0])
        valid_error_formatted = "{:.15f}".format(validation_error_list[i][0])
        print(f"Epoch {i+1}, Training Error: {train_error_formatted}, Validation Error: {valid_error_formatted}")
    return
```

Figure 13: Training Function Part 2

After training, the neural network undergoes testing using both training and test data to assess accuracy and identify misclassifications. In this process, each data point undergoes a forward pass, and the predicted output is compared with the true label. Accuracy is computed, and misclassifications for each class are recorded. Subsequently, the test results are visualized through plotting.



```
def test_accuracy(self, test_data, test_labels):
    total_accuracy = 0
    misclassified_counts = {i: 0 for i in range(1, 7)}  # Assuming class labels are 1-6

    for i in range(len(test_data) // self.batch_size):
        data_idx = test_data[self.batch_size * i: self.batch_size * (i + 1)]
        label_idx = test_labels[self.batch_size * i: self.batch_size * (i + 1)]
        one_hot_array = [[1 if i == idx-1 else 0 for i in range(6)] for idx in label_idx]
        one_hot_vector = np.array(one_hot_array)
        loss = self.forward(data_idx,one_hot_array)
        predicted_labels=self.o_3
        true_labels = np.argmax(one_hot_vector, axis=1)
        batch_predictions = np.argmax(predicted_labels, axis=1)

        correct_predictions = np.sum(batch_predictions == true_labels)
        batch_accuracy = correct_predictions / len(true_labels)
        total_accuracy += batch_accuracy

        # Update misclassification counts
        for label, prediction in zip(true_labels, batch_predictions):
            if label != prediction:
                misclassified_counts[label + 1] += 1  # Adjusting index for 1-6 range

    # Calculate overall accuracy
    overall_accuracy = total_accuracy / (len(test_data) // self.batch_size)
    print(f"Test Accuracy: {overall_accuracy * 100:.2f}%")

    # Print misclassification counts
    for label, count in misclassified_counts.items():
        print(f"Class {label}: {count} misclassifications")

    classes = list(misclassified_counts.keys())
    counts = [misclassified_counts[c] for c in classes]

    plt.bar(classes, counts, color='skyblue')
    plt.xlabel('Class Label')
    plt.ylabel('Number of Misclassifications')
    plt.title('Misclassifications per Class')
    plt.xticks(classes)
    plt.show()

    return
```

Figure 14: Test Accuracy Function

Ahmet Bera Özbolat
21902777

After implementing the neural network, the training phase is initiated for each of the 16 cases. During this training, accuracy and misclassifications are computed to evaluate the performance of each case, helping identify the most effective configuration.

TABLE I: Results Of Online Training

Case (n,N2,momentum)	Train MC	Test MC	Test Accuracy (%)	Training Accuracy (%)	# of Epoch	CPU Time (s)
0.001,100,0.0	46	206	92.67	99.37	100	1881
0.001,100,0.9	132	264	91.04	98.20	87	1631
0.001,200,0.0	44	208	92.94	99.40	100	1958
0.001,200,0.9	22	191	93.52	99.70	100	1976
0.01,100,0.0	55	184	93.76	99.25	41	744
0.01,100,0.9	127	239	91.89	98.27	47	858
0.01,200,0.0	66	195	93.38	99.10	49	967
0.01,200,0.9	83	273	90.74	98.75	45	884

TABLE II: Results Of Online Training (Misclassifactions)

Case (n,N2,momentum)	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6
0.001,100,0.0	12	47	31	82	17	27
0.001,100,0.9	7	62	27	137	5	26
0.001,200,0.0	10	36	39	80	16	27
0.001,200,0.9	6	36	38	66	18	27
0.01,100,0.0	6	38	40	64	23	12
0.01,100,0.9	13	43	37	105	13	28
0.01,200,0.0	7	44	42	65	28	9
0.01,200,0.9	8	70	62	87	17	29

TABLE III: Results Of Batch Training

Case (n,N2,momentum)	Train MC	Test MC	Test Accuracy (%)	Training Accuracy (%)	# of Epoch	CPU Time (s)
0.001,100,0.0	4354	1676	42.21	40.76	100	66
0.001,100,0.9	1872	836	71.17	74.53	100	68
0.001,200,0.0	3698	1366	52.90	49.69	100	71
0.001,200,0.9	1724	789	72.79	76.54	100	71
0.01,100,0.0	159	202	93.03	97.84	100	67
0.01,100,0.9	153	224	92.28	97.92	74	49
0.01,200,0.0	144	198	93.17	98.04	100	75
0.01,200,0.9	159	220	92.41	97.84	72	54

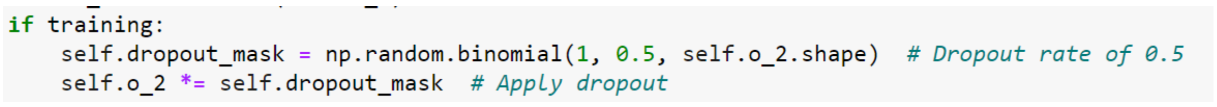
TABLE IV: Results Of Batch Training (Misclassifactions)

Case (n,N2,momentum)	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6
0.001,100,0.0	0	441	403	391	268	73
0.001,100,0.9	254	147	161	199	46	19
0.001,200,0.0	4	398	403	491	11	59
0.001,200,0.9	229	161	145	181	53	20
0.01,100,0.0	7	39	35	84	15	23
0.01,100,0.9	12	36	35	104	14	23
0.01,200,0.0	8	37	38	79	13	23
0.01,200,0.9	10	30	39	106	11	24

The training results reveal that the configuration with parameters n=0.01, N2=100, and momentum=0 consistently produces the best results. However, when selecting the optimal configuration among all options, online training with n=0.01, N2=100, and momentum=0 stands out as the superior choice. It's also worth noting that all configurations encounter difficulties in accurately identifying class 4, corresponding to the activity "Sitting."

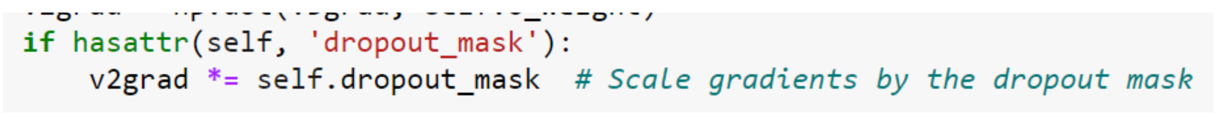
Dropout is a regularization technique employed in neural networks to combat overfitting. During training, it randomly deactivates a portion of neurons (typically expressed as a probability, like p=0.5) in each layer, forcing the network to become less reliant on specific neurons and encouraging the learning of more general features. This regularization prevents the neural network from closely memorizing the training data and enhances its ability to generalize to unseen data. By applying a dropout rate of p=0.5, you're deactivating each neuron with a 50% probability during training iterations, a commonly used rate that effectively controls overfitting. Incorporating dropout into your best-performing case with a dropout rate of p=0.5 enhances the model's robustness and promotes better generalization. During inference, dropout is typically turned off, allowing the full network capacity to make accurate predictions on new data[1].

To implement dropout in backpropagation, I made adjustments to both gradient calculation and forward propagation. Leveraging the 'np.random.binomial()' feature in Numpy, I created a mask array that is applied to the output of the second layer and the gradient of the second layer, effectively implementing the dropout technique. This mask array randomly deactivates a portion of neurons during training iterations, contributing to regularization and preventing overfitting in the neural network.



```
if training:
    self.dropout_mask = np.random.binomial(1, 0.5, self.o_2.shape)  # Dropout rate of 0.5
    self.o_2 *= self.dropout_mask  # Apply dropout
```

Figure 15: Drop Forward Modification



```
if hasattr(self, 'dropout_mask'):
    v2grad *= self.dropout_mask  # Scale gradients by the dropout mask
```

Figure 16: Dropout Gradient Modification

TABLE V: Results Of Best Case With Dropout

Case (n,N2,momentum)	Train MC	Test MC	Test Accuracy (%)	Training Accuracy (%)	# of Epoch	CPU Time (s)
0.01,100,0.0	57	180	93.89	99.22	76	1552

TABLE VI: Results Of Best Case With Dropout (Misclassifications)

Case (n,N2,momentum)	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6
0.011,100,0.0	11	40	44	58	17	10

[1] Dropout reduces underfitting - arxiv.org, accessed December 14, 2023, https://arxiv.org/pdf/2303.01500.pdf.

Conclusion

In this project, we successfully developed a feed-forward neural network for recognizing human activities using data from mobile device sensors. The project distinguished itself by its methodological approach, focusing on statistical features extracted from sensor data instead of direct time series analysis typically employed in recurrent neural networks (RNNs).

Through extensive experimentation with learning rates, momentum, mini-batch learning, and dropout techniques, we discovered that the optimal performance was achieved in the scenario where the second hidden layer had 100 neurons (N2 = 100), with a learning rate (η) of 0.01, no momentum, and in an online learning setting. This configuration resulted in the highest accuracy, surpassing the 90% threshold, which is a notable accomplishment for a basic neural network architecture in human activity recognition (HAR).

A key observation was that Class 4, representing a stationary activity, was the most frequently misclassified across all configurations. This could be attributed to the inherent challenges in distinguishing between subtle variations within stationary activities using sensor data.

The implementation of dropout, aimed at reducing overfitting and enhancing model performance, showed nuanced results. While dropout did not significantly improve the performance in the best-case scenario, it contributed to a slight increase in accuracy (around 0.1-0.2%) for the most effective configuration. This underscores the nuanced role of dropout in fine-tuning neural network models.

In many cases, early stopping effectively mitigated overfitting. However, when momentum and batch training were introduced, the early stopping mechanism was less effective, possibly due to missing the optimal stopping point. This highlights the delicate balance required in setting the parameters for early stopping, especially in the presence of momentum and batch training.

Overall, the project emphasized the efficacy of feature-based classification in HAR and the impact of various optimization techniques on model performance. The success with a simpler, feature-based approach over more complex RNNs provides valuable insights into efficient model design for HAR. The findings from this project offer a foundation for future exploration in HAR, with potential for further optimization and application in real-world scenarios.

APPENDIX

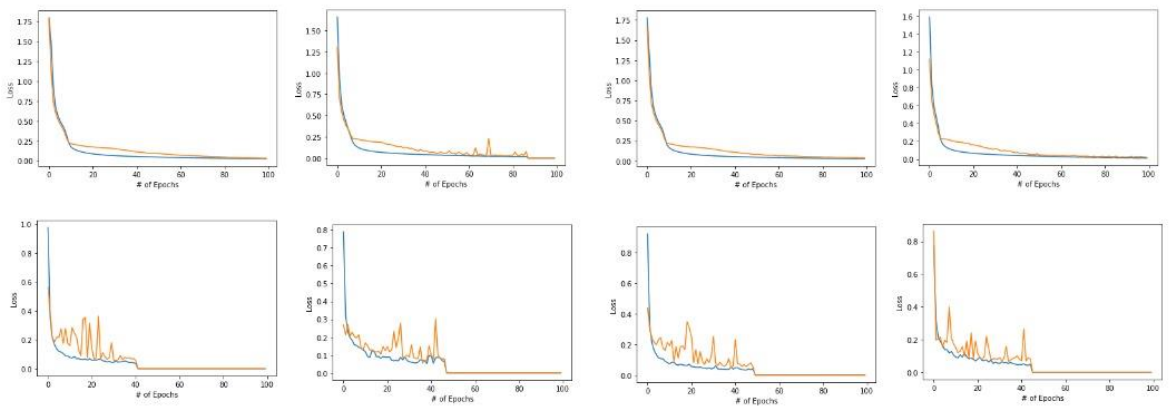


Figure 17: Online Training Error Plots (Same as the rankings in the table)

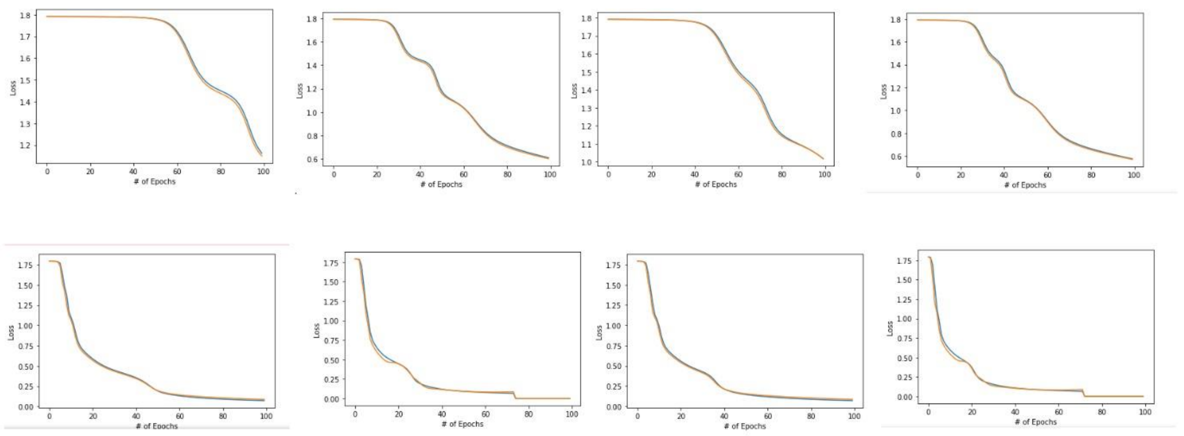


Figure 18: : Batch Training Error Plots (Same as the rankings in the table)

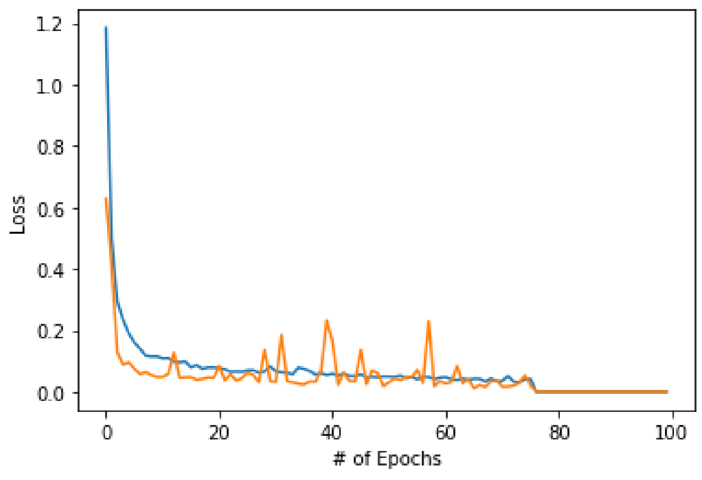


Figure 19: Best Case Dropout Error plot

REFERENCES

1-Dropout reduces underfitting - arxiv.org. Accessed December 14, 2023.
https://arxiv.org/pdf/2303.01500.pdf.