Assignment 1

Machine Learning 1, SS24

Team Members		
Last name	First name	Matriculation Number
Nožić	Naida	12336462
Hinum-Wagner	Jakob	01430617

1 Neural Networks [15 points]

In this task, we will use an implementation of Multilayer Perceptron (MLP) from `scikit-learn`. The documentation for this is available at the scikit-learn website. The relevant multi-layer perceptron class is `MLPClassifier`, as we will solve a classification task.

1.1 PCA and Classification [12 points]

PCA can be used as a data preprocessing technique, to reduce the dimensionality of data. In this task, we will use the Sign Language dataset. It consists of images of hands that should be classified into 10 different classes, as shown in Fig. 3. The images are of size $(64, 64)$ pixels, i.e., the input dimension to the neural network that we want to train to classify the images would be quite large ($64 \cdot 64 = 4096$), and hence we want to reduce their dimension by means of PCA. By doing this, the benefit will be that the model will be smaller and thus, it will take less time to train the model.

Tasks:

1.2 Task 1:Problem

PCA for dimensionality reduction. Use `PCA` from `sklearn.decomposition` to reduce the dimensionality of the training data `X_train`. Create an instance of `PCA` class, and set `random_state` to 42. Choose `n_components` (number of principal components) such that about 95% of variance is explained. In the report, state `n_components` that you used, and the exact percentage of variance explained that you got.

1.3 Task 1:Solution

The Sign Language dataset consists of images of hands classified into 10 different classes. Each image is of size 64x64 pixels, resulting in an input dimension of 4096 (64×64) for a neural network. Training a model with such high-dimensional data can be computationally expensive and time-consuming. Therefore, we employ Principal Component Analysis (PCA) to reduce the dimensionality of the training data, making the model smaller and faster to train.

Explanation of the Code

Standardization The first step is to standardize the training data $\mathbf{X}_{\text{train}}$ to ensure that each feature contributes equally to the PCA. This is achieved using `StandardScaler`, which scales the data to have a mean of 0 and a standard deviation of 1.

PCA Initialization We create an instance of the `PCA` class with `n_components` set to 0.95 and `random_state` set to 42 for reproducibility. Setting `n_components` to 0.95 means that PCA will select the number of principal components required to retain 95% of the variance in the data.

Fitting and Transforming The PCA model is then fitted to the standardized data, and the data is transformed to the new reduced dimensionality space. The transformed data $\mathbf{X}_{\text{train\_pca}}$ has a reduced number of features.

Explained Variance and Components The code prints the total explained variance ratio and the number of principal components selected. This is calculated using $\sum$(pca.explained_variance_ratio_).This information is crucial to verify that the desired amount of variance is retained.

Results

Initially, the dataset dimensions were as follows:

- Features shape: (2062, 4096)

- Targets shape: (2062,)

After applying PCA to the Sign Language dataset, we find that:

- **Number of Components Used**: The number of principal components selected by setting `n_components=0.95` was automatically determined by PCA to be 301.
- **Exact Percentage of Variance Explained**: The exact percentage of variance explained by the selected components was calculated using $\sum$(pca.explained_variance_ratio_) and found to be 95.01% (0.9501413419930032).

The transformed training data had the following shape:

- Transformed training data shape: (1649, 301)

By reducing the dimensionality of the training data using PCA, we significantly decreased the computational load and training time of the neural network model, while still retaining most of the important information from the original high-dimensional data. This step is crucial in efficiently handling large datasets and improving model performance.

1.4 Task 2:Problem

Varying the number of hidden neurons. We will use the data with the reduced dimension (from the previous step) to train a neural network to perform classification. We will vary the number of neurons in one hidden layer of the neural network: `n_hidden` $\in \{2, 10, 100, 200, 500\}$.

For this task, we will use `MLPClassifier` from `sklearn.neural_network`. Create an instance of `MLPClassifier`. Set `max_iter` to 500, `solver` to `adam`, `random_state` to 1, `hidden_layer_sizes` to be `n_hidden` (a value in the set above), and the other parameters should have their default values. (If the warning about the optimization not converging appears, you can ignore it. No need to do anything about it.)

For each `num_hidden` $\in \{2, 10, 100, 200, 500\}$, report the accuracy on the train and validation set, and the final training loss.

1.5 Task 2: Solution

Explanation of the Code

Function Definition The function `train_nn` trains an `MLPClassifier` with different numbers of neurons in a single hidden layer.

Data Splitting The function starts by splitting the training data into training and validation sets using `train_test_split` with 20% of the data as the validation set. The validation set is used to evaluate the model's performance on unseen data, helping to avoid overfitting and to select the best model configuration.

Training the Neural Network We then define a list of different values for the number of hidden neurons: `[2, 10, 100, 200, 500]`. The function iterates over these values, trains an `MLPClassifier` with the specified number of hidden neurons, and evaluates its performance. The `MLPClassifier` is instantiated with `max_iter` set to 500, `solver` set to 'adam', and `random_state` set to 1. For each configuration, the training accuracy, validation accuracy, and final training loss are printed.

Training Accuracy, Validation Accuracy, and Final Training Loss - **Training Accuracy**: This is the accuracy of the model on the training set. It indicates how well the model has learned the training data. - **Validation Accuracy**: This is the accuracy of the model on the validation set. It is crucial for evaluating how well the model generalizes to unseen data. - **Final Training Loss**: This is the loss value at the end of the training process. It indicates how well the model has minimized the error during training.

The best configuration is defined by the highest validation accuracy, as it indicates the model's ability to generalize to new, unseen data.

Results

The results for each configuration of hidden neurons are as follows:

- **2 hidden neurons**:
  - Training accuracy: 0.4996
  - Validation accuracy: 0.2667
  - Final training loss: 1.3606
- **10 hidden neurons**:
  - Training accuracy: 0.9985
  - Validation accuracy: 0.6909
  - Final training loss: 0.0174
- **100 hidden neurons**:
  - Training accuracy: 1.0
  - Validation accuracy: 0.7545
  - Final training loss: 0.0058
- **200 hidden neurons**:
  - Training accuracy: 1.0
  - Validation accuracy: 0.7727
  - Final training loss: 0.0047
- **500 hidden neurons**:
  - Training accuracy: 1.0
  - Validation accuracy: 0.7606
  - Final training loss: 0.0036

From these results, we can observe that increasing the number of hidden neurons generally improves the validation accuracy up to a certain point. The best validation accuracy was achieved with 200 hidden neurons, making it the optimal choice for this dataset and task. The best configuration is defined by the highest validation accuracy, as it indicates the model's ability to generalize to new, unseen data.

1.6 Task 3: Problem

In general: How do we know if a model does not have enough capacity (the case of underfitting)? How do we know if a model starts to overfit?

In your particular case: Does overfitting/underfitting happen with some of your architectures/models? (If so, say with what number of neurons that happens). Which of the trained models would you perfer? Explain why.

1.7 Task 4: Solution

Underfitting

Underfitting occurs when a model is too simple to capture the underlying patterns in the data. This can be identified by low training accuracy and low validation accuracy. The model does not perform well on the training data, indicating it has not learned the relationships in the data adequately. In terms of model capacity, an underfitting model has insufficient capacity, meaning it lacks the complexity to learn the intricate patterns present in the data. This often happens with models that have too few parameters or features, as they cannot represent the underlying structure of the data.

Overfitting

Overfitting happens when a model is too complex and captures noise in the training data as if it were a pattern. This can be identified by a high training accuracy but a low validation accuracy. The model performs well on the training data but poorly on the validation data, indicating it does not generalize well to unseen data. In terms of model capacity, an overfitting model has excessive capacity, meaning it is too flexible and can memorize the training data, including the noise, rather than learning the general patterns. This often happens with models that have too many parameters or features, as they can fit the training data too closely.

Case Analysis of Overfitting/Underfitting in Our Models

In our experiments with varying the number of hidden neurons, we observed the following:

- With **2 hidden neurons**, the model exhibited underfitting, as indicated by both low training accuracy (0.4996) and low validation accuracy (0.2667). This suggests that the model was too simple to capture the underlying patterns in the data, demonstrating insufficient capacity.
- With **10 hidden neurons**, the model's training accuracy was very high (0.9985), but the validation accuracy was moderate (0.6909). This could be an indication of the model starting to overfit, but not excessively. The model has increased capacity compared to the 2-neuron configuration, but it may still be inadequate for capturing all the necessary patterns without overfitting.
- With **100 hidden neurons**, the training accuracy reached 1.0, and the validation accuracy improved to 0.7545, suggesting the model was learning well and generalizing better. The model's capacity appears to be more appropriate for the complexity of the data.
- With **200 hidden neurons**, we achieved the highest validation accuracy (0.7727) with a training accuracy of 1.0. This indicates that the model had sufficient capacity to learn the patterns in the training data and generalize well to the validation data. The capacity of the model seems well-matched to the data complexity.
- With **500 hidden neurons**, while the training accuracy remained 1.0, the validation accuracy slightly decreased to 0.7606. This suggests the model might be starting to overfit, capturing noise in the training data. The model's capacity might be becoming too high, leading to overfitting.

Preferred Model and Justification

The model with **200 hidden neurons** is preferred. This configuration achieved the highest validation accuracy (0.7727) while maintaining a perfect training accuracy (1.0). This indicates that the model had the right balance of complexity, allowing it to learn the training data well without overfitting to noise. The slight decrease in validation accuracy with 500 hidden neurons suggests that increasing complexity beyond this point starts to capture noise, making the model less generalizable. Therefore, the model with 200 hidden neurons has sufficient capacity to accurately represent the data without overfitting, making it the optimal choice for this dataset and task.

1.8 Task 5:Problem

Overfitting. To prevent overfitting, we could use a few approaches, for example, introducing regularization and/or early stopping. Copy the code from the previous task. Try out (a) $\alpha = 0.1$ (b) `early_stopping = True`, (c) $\alpha = 0.1$ and `early_stopping = True`. Choose (a), (b), or (c) - depending on what you think works best. State your choice in your report.

Then, using a setup as in (a), (b), or (c), report the train and validation accuracy, and the loss for `num_hidden` $\in \{2, 10, 100, 200, 500\}$. Does this improve the results from the previous step? Which model would you choose now?

1.9 Task 5: Solution

To prevent overfitting, we experimented with three approaches: 1. Using regularization with $\alpha = 0.1$. 2. Enabling early stopping. 3. Combining regularization ($\alpha = 0.1$) and early stopping.

Function Definition

The function `train_nn_with_regularization` modifies the previous `train_nn` function by incorporating regularization and early stopping.

Configurations

We tested the following configurations:

- $\alpha = 0.1$, early stopping = False
- $\alpha = 0.0001$, early stopping = True
- $\alpha = 0.1$, early stopping = True

Results

The results for each configuration and number of hidden neurons are as follows:
Configuration: $\alpha = 0.1$, early_stopping = False

- **2 hidden neurons**:
  - Training accuracy: 0.5163
  - Validation accuracy: 0.3515
  - Final training loss: 1.3803
- **10 hidden neurons**:
  - Training accuracy: 0.9962
  - Validation accuracy: 0.7121
  - Final training loss: 0.1036
- **100 hidden neurons**:
  - Training accuracy: 1.0
  - Validation accuracy: 0.7848
  - Final training loss: 0.0664
- **200 hidden neurons**:
  - Training accuracy: 1.0
  - Validation accuracy: 0.8030
  - Final training loss: 0.0634
- **500 hidden neurons**:
  - Training accuracy: 1.0
  - Validation accuracy: 0.8061
  - Final training loss: 0.0615

Configuration: $\alpha = 0.0001$, early_stopping = True

- **2 hidden neurons**:

  - Training accuracy: 0.1744
  - Validation accuracy: 0.1545
  - Final training loss: 2.0758

- **10 hidden neurons**:
  - Training accuracy: 0.8165
  - Validation accuracy: 0.6515
  - Final training loss: 0.6309

- **100 hidden neurons**:
  - Training accuracy: 0.9522
  - Validation accuracy: 0.7273
  - Final training loss: 0.1452

- **200 hidden neurons**:
  - Training accuracy: 0.9659
  - Validation accuracy: 0.7515
  - Final training loss: 0.0967

- **500 hidden neurons**:
  - Training accuracy: 0.9765
  - Validation accuracy: 0.7727
  - Final training loss: 0.0165

Configuration: $\alpha = 0.1$, early_stopping = True

- **2 hidden neurons**:
  - Training accuracy: 0.1744
  - Validation accuracy: 0.1545
  - Final training loss: 2.0786

- **10 hidden neurons**:
  - Training accuracy: 0.8575
  - Validation accuracy: 0.6727
  - Final training loss: 0.5336

- **100 hidden neurons**:
  - Training accuracy: 0.9538
  - Validation accuracy: 0.7303
  - Final training loss: 0.2082

- **200 hidden neurons**:
  - Training accuracy: 0.9765
  - Validation accuracy: 0.7576
  - Final training loss: 0.1380

- **500 hidden neurons**:
  - Training accuracy: 0.9773
  - Validation accuracy: 0.7788
  - Final training loss: 0.1352

Best Configuration

The best configuration to prevent overfitting is using regularization with $\alpha = 0.1$ and no early stopping, achieving a validation accuracy of 0.8061 with 500 hidden neurons. This configuration provides the best balance between training accuracy, validation accuracy, and final training loss, indicating a well-regularized model that generalizes well to unseen data.

1.10 Task 6:Problem

From all models you have trained so far, pick the one you consider best. For this model, plot the loss curve (training loss over iterations). Hint: Check the attributes of the classifier.

1.11 Task 6:Solution

The best configuration to prevent overfitting is using regularization with $\alpha = 0.1$ and no early stopping, achieving a validation accuracy of 0.8061 with 500 hidden neurons. This configuration provides the best balance between training accuracy, validation accuracy, and final training loss, indicating a well-regularized model that generalizes well to unseen data.

Best Model

After evaluating all trained models, the best model is the one trained with regularization $\alpha = 0.1$ and no early stopping, with 500 hidden neurons. This model achieved:

- Training accuracy: 1.0
- Validation accuracy: 0.8061
- Final training loss: 0.0615

This model is preferred because it demonstrates the highest validation accuracy, indicating its superior ability to generalize to new, unseen data. Additionally, the final training loss shows that the model effectively minimized the error during training, further validating its robustness and reliability. However, it's important to note that an excessively high final training loss can indicate underfitting or issues with the training process, so it should be monitored and minimized as much as possible while maintaining good validation performance.

Training Loss Curve To further analyze the performance of the best model, we plot the training loss curve, which shows the loss over iterations during the training process. This curve helps to understand how well the model is learning over time.

The training loss curve indicates a rapid decrease in loss initially, which slows down and converges as training progresses. This suggests that the model is effectively learning and minimizing the loss function. The final low loss value indicates good model performance and proper convergence, validating the choice of the model configuration.

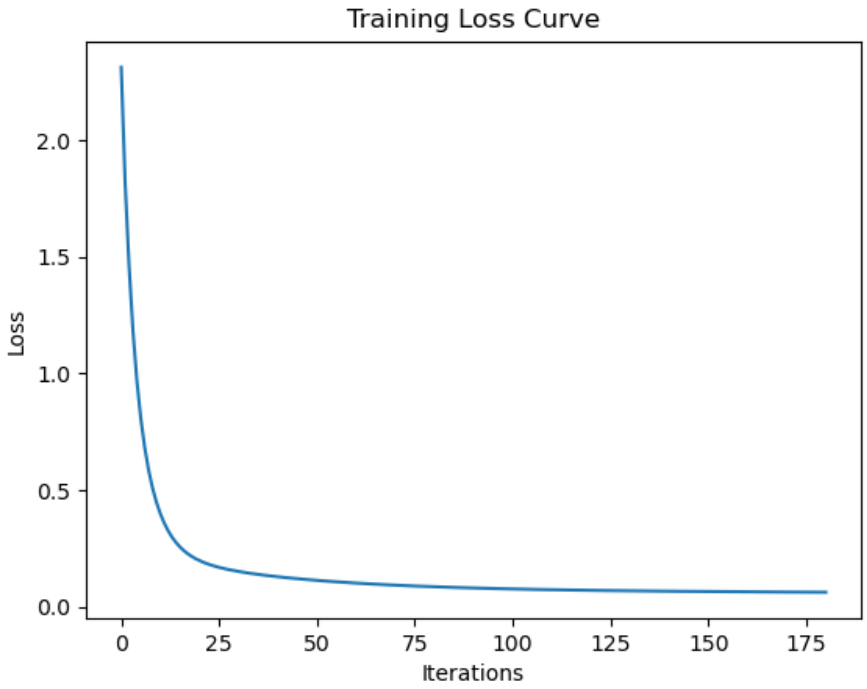

Figure 1: Training Loss Curve

1.12 Model selection and Evaluation Metrics [3 points]

To find a good configuration of hyperparameters, we can use, for example, `GridSearchCV`, by trying out all the different combinations of the parameters by an automated procedure.

Tasks:

1.13 Task 7: Problem

We want to check all possible combinations of the parameters:

- $\alpha \in \{0.0, 0.1, 1.0\}$
- `solver` $\in$ { `'lbfgs', 'adam'` }
- `hidden_layer_sizes` $\in \{(100,), (200,)\}$

Create a dictionary of these parameters that `GridSearchCV` from `sklearn.model_selection` requires. How many different architectures will be checked? (State the number of architectures that will be checked and how you calculated it.)

1.14 Task 7: Solution

To find a good configuration of hyperparameters, we use `GridSearchCV`, which tries out all the different combinations of the parameters by an automated procedure. We want to check all possible combinations of the parameters:

- $\alpha \in \{0.0, 0.1, 1.0\}$
- `solver` $\in \{$`'lbfgs', 'adam'`$\}$
- `hidden_layer_sizes` $\in \{(100,), (200,)\}$

Create a dictionary of these parameters that `GridSearchCV` from `sklearn.model_selection` requires. The number of different architectures that will be checked is calculated by multiplying the number of choices for each parameter: $3 \times 2 \times 2 = 12$. Hence, 12 different architectures will be checked.

Task 8: Problem

Set `max_iter=100, random_state=42` as default parameters of `MLPClassifier`. Create a `GridSearchCV` object with the dictionary you have created. Run cross validation with $k = 5$ folds by setting `cv=5`. If you want a more verbose output during the search procedure, set e.g. `verbose=4`.

Task 8: Solution

Explanation of the Code

Parameter Dictionary We create a dictionary of hyperparameters to be used in `GridSearchCV`. The parameters to be tested are:

- $\alpha \in \{0.0, 0.1, 1.0\}$
- `solver` $\in \{$`'lbfgs', 'adam'`$\}$
- `hidden_layer_sizes` $\in \{(100,), (200,)\}$

MLPClassifier Initialization We initialize an `MLPClassifier` with the default parameters, setting `max_iter=100` and `random_state=42`.

GridSearchCV Setup We create a `GridSearchCV` object with the parameter grid, setting `cv=5` for 5-fold cross-validation and `verbose=4` for more detailed output during the search procedure.

Running GridSearchCV We fit the `GridSearchCV` object to the training data. `GridSearchCV` will evaluate all 12 different configurations of the hyperparameters and select the one with the best performance based on cross-validation.

Best Model Selection The best model found by `GridSearchCV` is printed along with its cross-validation score and parameters.

```
def perform_grid_search(X_train: np.ndarray, y_train: np.ndarray) -> MLPClassifier:
    """
    Perform GridSearch using GridSearchCV.

    :param X_train: PCA-projected features with shape (n_samples, n_components)
    :param y_train: Targets
    :return: The best estimator (MLPClassifier) found by GridSearchCV
    """
    param_grid = {
        'alpha': [0, 0.1, 1.0],
        'solver': ['lbfgs', 'adam'],
        'hidden_layer_sizes': [(100,), (200,)]
    }

    mlp = MLPClassifier(max_iter=100, random_state=42)

    grid_search = GridSearchCV(estimator=mlp, param_grid=param_grid, cv=5, verbose=4)

    grid_search.fit(X_train, y_train)

    print(f"Best score: {grid_search.best_score_}")
    print(f"Best parameters: {grid_search.best_params_}")

    return grid_search.best_estimator_
```

The `GridSearchCV` will evaluate 12 different configurations of the hyperparameters and select the one with the best performance based on cross-validation.

Task 8: Problem

What was the best parameter set that was found in this grid search? What was the best score obtained (i.e., the mean cross-validation score)? Hint: Check the attributes of the `GridSearchCV` object.

1.15 Task 8: Solution

The best parameter set found in this grid search was:

- α: 1.0
- `hidden_layer_sizes`: (200,)
- `solver`: 'lbfgs'

The best mean cross-validation score obtained was 0.8338.

Comparing Mean CV Score to Validation Accuracy

The mean cross-validation (CV) score obtained from GridSearchCV represents the average performance of the model across multiple folds of the data. This score is generally considered more robust and reliable compared to the validation accuracy obtained from a single train-validation split, as it reduces the variance associated with any one split of the data.

Both metrics aim to measure the model's ability to generalize to new, unseen data, and hence they are inherently comparable. The mean CV score typically provides a better estimate of the model's generalization performance, while the single validation accuracy can be influenced by the specific choice of training and validation sets.

In conclusion, while both the mean CV score and the validation accuracy are important, the mean CV score is generally preferred for its robustness and reliability. If the mean CV score and the validation accuracy are close, it confirms the model's stability. If they differ significantly, the mean CV score should generally be trusted more for making the final decision.

1.16 Task 9:Problem

Implement `show_confusion_matrix_and_classification_report`. Among all models trained so far (also the ones you have constructed without the help of `GridSearchCV`), pick the model you consider best. Then, pass this final model to the implemented function (together with the PCA-projected test set). Report the final test accuracy in your report. Also include the plot of the confusion matrix and the classification report.

1.17 Task 9: Solution

To evaluate the best model selected so far, we implement the function `show_confusion_matrix_and_classification_report` to plot the confusion matrix and print the classification report.

```
def show_confusion_matrix_and_classification_report(nn: MLPClassifier, X_test: np.ndarray, y_test: np.n
    """
    Plot confusion matrix and print classification report.

    :param nn: The trained MLPClassifier you want to evaluate
    :param X_test: Test features (PCA-projected)
    :param y_test: Test targets
    """
    y_pred = nn.predict(X_test)

    # Compute the confusion matrix
    cm = confusion_matrix(y_test, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm)

    # Plot the confusion matrix
    disp.plot(cmap=plt.cm.Blues)
    plt.title("Confusion Matrix")
    plt.show()

    # Print the classification report
    report = classification_report(y_test, y_pred)
    print("Classification Report:")
    print(report)
```

After selecting the best model, we pass this model to the function along with the PCA-projected test set. The final test accuracy, confusion matrix, and classification report are reported below.

- **Test Accuracy**: 0.87

The confusion matrix discussion is done in the next task.

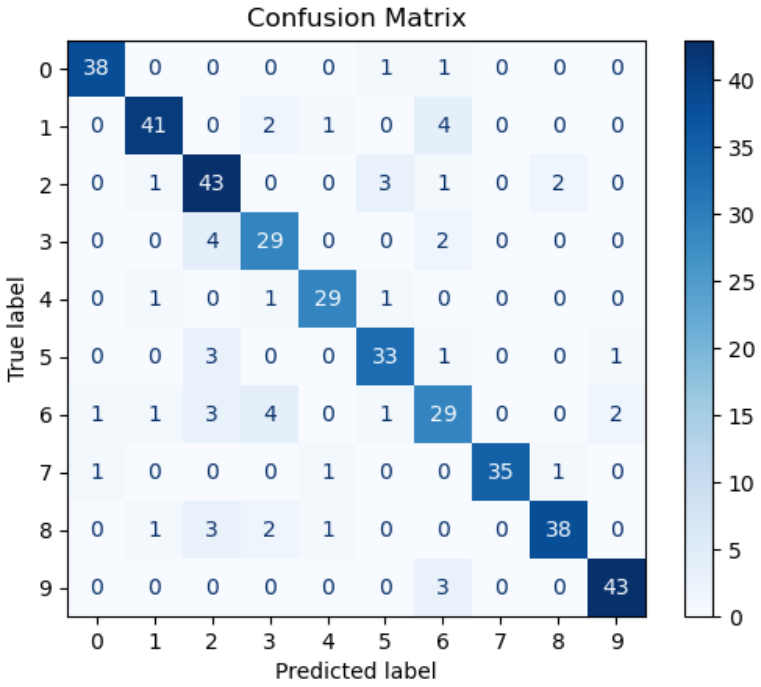

Figure 2: Confusion Matrix

Classification Report

```
Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.95      0.95        40
           1       0.91      0.85      0.88        48
           2       0.77      0.86      0.81        50
           3       0.76      0.83      0.79        35
           4       0.91      0.91      0.91        32
           5       0.85      0.87      0.86        38
           6       0.71      0.71      0.71        41
           7       1.00      0.92      0.96        38
           8       0.93      0.84      0.88        45
           9       0.93      0.93      0.93        46

    accuracy                           0.87       413
   macro avg       0.87      0.87      0.87       413
weighted avg       0.87      0.87      0.87       413
```

The confusion matrix and classification report indicate that the model performs well across all classes, with an overall accuracy of 0.87. This validates the model's effectiveness in classifying the sign language images.

1.18 Task 10:Problem

Explain in words what *recall* measures and what *precision* measures. Which class in the test dataset was misclassified most often?

1.19 Task 10: Solution

Precision

Precision measures the accuracy of positive predictions. It is the ratio of true positive predictions to the total number of positive predictions (true positives + false positives). Precision answers the question: "Of all the instances that were predicted as positive, how many were actually positive?"

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Recall

Recall (also known as Sensitivity or True Positive Rate) measures the ability of the model to find all the positive instances. It is the ratio of true positive predictions to the total number of actual positives (true positives + false negatives). Recall answers the question: "Of all the actual positive instances, how many were correctly predicted as positive?"

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

Misclassified Class Analysis

To identify which class was misclassified most often, we analyze the confusion matrix. The class with the highest number of misclassifications (both false negatives and false positives) is considered the most often misclassified class.

Analysis From the confusion matrix, we observe the following misclassifications for each class:

- **Class 0**: 2 misclassifications
- **Class 1**: 7 misclassifications
- **Class 2**: 7 misclassifications
- **Class 3**: 10 misclassifications
- **Class 4**: 3 misclassifications
- **Class 5**: 5 misclassifications
- **Class 6**: 12 misclassifications
- **Class 7**: 2 misclassifications
- **Class 8**: 7 misclassifications
- **Class 9**: 3 misclassifications

The class with the most misclassifications is **class 6** (True label), which has:

- 1 false positive predicted as class 0
- 3 false positives predicted as class 1
- 4 false positives predicted as class 2
- 1 false positive predicted as class 4
- 1 false positive predicted as class 5
- 2 false positives predicted as class 9

This indicates that class 6 was misclassified as other classes a total of 12 times (1 + 3 + 4 + 1 + 1 + 2 = 12), making it the most often misclassified class.

Conclusion

- **Precision** measures how many of the predicted positive instances are actually positive.
- **Recall** measures how many of the actual positive instances were correctly predicted.
- The class that was misclassified most often in the test dataset is **class 6**.

1.20 Task 11: Problem

(Theoretical question, general) What is the difference between hyperparameters and parameters of a model (in general)? Explain then the difference using the example of neural networks (i.e., name a few hyperparameters and parameters of neural networks).

1.21 Task 11: Solution

General Definitions

In the context of machine learning, it is crucial to distinguish between *hyperparameters* and *parameters*:

Parameters *Parameters* are internal variables of a model that are learned from the training data during the training process. These values are adjusted by the learning algorithm to minimize the error on the training set. Parameters directly influence the model's predictions.

Examples of Parameters:

- The weights in a neural network
- The coefficients in a linear regression model
- The support vectors in a support vector machine

Hyperparameters *Hyperparameters*, on the other hand, are external settings to the model that are specified prior to the training process. They govern the training process and the structure of the model but are not updated during training. Hyperparameters are set before the learning process begins and are often tuned using techniques such as grid search or random search to find the best values for a given problem.

Examples of Hyperparameters:

- The learning rate in gradient descent
- The number of hidden layers and neurons in a neural network
- The regularization parameter (e.g., α in Lasso regression)
- The maximum depth of a decision tree

Example: Neural Networks

In the context of neural networks, the distinction between parameters and hyperparameters is particularly clear:

Parameters in Neural Networks

- **Weights**: These are the coefficients that are applied to the input features and are adjusted during training through backpropagation to minimize the loss function.
- **Biases**: These are additional parameters added to each neuron that are also learned during the training process to help the model fit the data better.

Hyperparameters in Neural Networks

- **Learning Rate**: Controls how much the model's parameters are adjusted with respect to the loss gradient during each update.
- **Number of Hidden Layers**: Determines how many layers of neurons are present between the input and output layers.
- **Number of Neurons per Layer**: Specifies the number of neurons in each hidden layer.
- **Batch Size**: Defines the number of training examples used in one iteration of the training process.
- **Epochs**: The number of times the entire training dataset is passed through the network.
- **Regularization Parameters**: Such as dropout rate, L_2 regularization factor, etc., which are used to prevent overfitting.

2 Neural Networks From Scratch [10 points, 1* bonus point]

In this task, we will implement a neural network from scratch, i.e., without using `MLPClassifier` from `scikit-learn`. We will again solve the same classification task as above.

In order to train the network, we will use a minimal implementation of an *automatic differentiation* framework. This will allow us the compute gradients of the loss w.r.t. the weights and biases of the neural network, which in turn allows us to perform (stochastic) gradient descent.

The central component of this framework is the `Scalar` class in the `autodiff` module. An object of this class can store a single scalar value $x \in \mathbb{R}$ and, as soon as we compute derivatives of the loss $\mathcal{L}$, this object will also contain the partial derivative $\frac{\partial \mathcal{L}}{\partial x}$ at the point x (which is again just a scalar value). The `Scalar` class keeps track of a computational graph by overloading methods like `__add__` or `__mul__` (which are called when you add or multiply two Python objects, respectively). Thus, we need to implement all computations we wish to differentiate using `Scalar` objects (i.e., the neural network forward pass, including all activation functions, and the loss function).

For example, if we wish to add two Scalar objects `s1 = Scalar(1.0)` and `s2 = Scalar(2.0)`, we just compute `s = s1 + s2`, where `s` is again a `Scalar` object which holds the result of the computation, as well as the computational graph that produced this result.

Tasks:

2.1 Task 1: Problem

In `autodiff/neural_net.py`, implement the `__call__` method of the `Neuron` class. This method should compute the output of a single neuron, given a list of inputs. Note that the inputs are given as a list of `Scalar` objects and you should again return a `Scalar` object.

Task 1: Solution

A single neuron is a building block of every neural network and each one individually performs a simple computation given by the following formula:

$$\hat{y} = \theta(\sum_{i=0}^{D}(w_i x_i))$$

- D is the number of inputs
- x_i represents the inputs, which can be features or outputs of previous neurons. In our code the inputs are given as a parameter to the function `__call__`.
- w_i represents the weights and each neuron object has them saved as one of its attributes. If a neuron is newly created, the starting weights are initialized in the `__init__` method.
- w_0 is the bias with $x_0 = 0$.
- θ is the activation function applied to the output of a neuron.

This exact formula is modelled in the code within `__call__`. After computing the defined sum, we apply the ReLu function depending on whether the corresponding attribute is set to true or not.

2.2 Task 2: Problem

In `autodiff/neural_net.py`, implement the `__init__` and `__call__` method of the `FeedForwardLayer` class.

Task 2: Solution

In the `__init__` we initialize the `FeedForwardLayer`, which will contain a list of neurons. Each neuron is created based on the number of inputs and whether or not a ReLu function will be applied to its output. Additionally, the number of neurons in the layer is determined by the `num_outputs` parameter.

On the other hand, the `__call__` method obtains the output of all neurons in the forward layer. If we analyze it from the code perspective, we are simply calling the `__call__` method on each neuron in the previously created list.

2.3 Task 3: Problem

In `autodiff/neural_net.py`, implement the `__init__` and `__call__` method of the `MultiLayerPerceptron` class

Task 3: Solution

A MultiLayerPerceptron is a neural network, which consists of multiple layers. It can have one input and output layer with multiple hidden layers in between. The `__init__` method is implemented in a way so it initializes all of the mentioned layers and then iterates through them while creating a corresponding `FeedForwardLayer` object. It is also important to note that the ReLu function will only be applied to hidden layers. The task of the `__call__` method in MultiLayerPerceptron is simple. We just iterate though each layer and call its corresponding `__call__` method.

2.4 Task 4: Problem

In `mlp_classifier_own.py`, you can find an implementation of a multi-class[1] classifier that uses your implementation of the MultiLayerPerceptron class under the hood. Carefully read and understand the code in this file. Note that when the MLP in `self.model` performs a forward pass, we get the *logits* at the output layer, i.e., the values that still need to be fed into the output activation function. Implement the `softmax` method and the `multiclass_cross_entropy_loss` method.

Task 4: Solution

For this task we have simply implemented the formula for calculating the Softmax and Multiclass Cross Entropy Loss in the code. The formulas are defined below:

$$softmax(x_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

$$MulticlassCrossEntropyLoss(y, \hat{y}) = -\log(\hat{y}_i)$$

In the code, when calculating the Softmax function value, I have always subtracted the maximum value element from the current one before exponentiating in the step e^{z_i}. This is done to ensure that the exponentiated value does not become excessively large and cause overflow.

[1] We will extend this to also work with binary classification in Task 3, so you can ignore any code that is run when `self.num_classes == 2` for now.

2.5 Task 5: Problem

In `nn_classification_from_scratch.py`, implement the `train_nn_own` function. In there, create an `MLPClassifierOwn` object: We want to train for 5 epochs, set the L2 regularization coefficient $\alpha = 0$, construct a network with a single hidden layer with 16 neurons and set `random_state=42`. As input to this small network we will use the PCA-projected *Sign Language* dataset – however, this time, we will only use 16 principle components for the PCA projection (`n_components=16`). We take these measures to keep the network size small, as our implementation is highly inefficient (no parallelization/vectorization). After training, use the `score` method of the classifier to compute the accuracy on the train, validation and test set. Report the final train, validation and test accuracy in your report.

Task 5: Solution

The final accuracy values for $\alpha = 0$ are given below:

- Test accuracy: 0.6586
- Train accuracy: 0.7566
- Validation accuracy: 0.6636

2.6 Bonus task : Problem

Bonus tasks [1* bonus point]: Right now, we can only train our network without L2 regularization ($\alpha = 0$). Implement the `l2_regularization_term` method in `mlp_classifier_own.py`. The output of this method is added to the loss during training. Specifically, let $\Omega(\boldsymbol{\theta})$ denote the output of this method. Then we have

$$\Omega(\boldsymbol{\theta}) = \frac{\alpha}{2|\mathcal{B}|}\|\boldsymbol{\theta}\|_2^2$$

where $\boldsymbol{\theta}$ is the vector of network parameters and $|\mathcal{B}|$ is the batch size.

Train the same network in Task 2, but now set $\alpha > 0$. Pick two different values for α and see if you can observe any difference to the case where $\alpha = 0$. State which value of α you have tried. Do you think that L2 regularization is useful in this case?

Bonus task: Solution

In the `l2_regularization_term` method we have implemented the

$$\Omega(\boldsymbol{\theta}) = \frac{\alpha}{2|\mathcal{B}|}\|\boldsymbol{\theta}\|_2^2$$

formula. For obtaining the results, we have tested out $\alpha = 0.5$, after which we have obtained:

- Test accuracy: 0.6780
- Train accuracy: 0.721
- Validation accuracy: 0.6576

For $\alpha = 1.0$ we got:

- Test accuracy: 0.6610
- Train accuracy: 0.6892
- Validation accuracy: 0.6273

The results are similar to the first case with $\alpha = 0.0$. However the biggest difference can be seen in the batch loss values,which are significantly higher. In general, the l2 regularization term penalizes large weights in the model. By adding it to the loss we can prevent the function from overfitting. What overfitting usually does is make the loss too low for the training data but as a result the model performs worse during the validation. Therefore, it is expected for the loss values to increase. However, using l2 regulaziration has not been significantly useful in this case because the accuracies obtained have not improved much compared to before.

3 Bonus: Binary Classification Extension [4* bonus points]

In this bonus task, we wish to extend `MLPClassifierOwn` to not only support multi-class classification, but also *binary classification* (i.e., `num_classes == 2`).

Tasks:

3.1 Task 1: Problem

In `mlp_classifier_own.py`, implement the methods `sigmoid` and `binary_cross_entropy_loss`.

3.2 Task 1: Solution

- In `mlp_classifier_own.py`, implement the `sigmoid` method as a static method to compute the sigmoid of a given `Scalar`. The `sigmoid` function is defined as:

```
@staticmethod
def sigmoid(z: Scalar) -> Scalar:
    """
    Returns the sigmoid of the given Scalar (as another Scalar).

    :param z: Scalar
    """
    return 1 / (1 + (-z).exp())
```

  This method takes a `Scalar` object `z` and returns its sigmoid, which is useful for converting the output of the neural network into a probability value between 0 and 1.

- In `mlp_classifier_own.py`, implement the `binary_cross_entropy_loss` method as a static method to compute the binary cross-entropy loss for a single sample. The binary cross-entropy loss function is defined as:

```
@staticmethod
def binary_cross_entropy_loss(y_true: int, prob: Scalar) -> Scalar:
    """
    Returns the binary cross-entropy loss for a single sample.

    :param y_true: 0 or 1
    :param prob: Scalar between 0 and 1, representing the probability of the positive class
    """
    return -y_true * prob.log() - (1 - y_true) * (1 - prob).log()
```

  This method takes an integer `y_true` (which is either 0 or 1) and a `Scalar` object `prob` (representing the predicted probability of the positive class), and returns the binary cross-entropy loss as a `Scalar`.

By implementing these methods, we can extend the functionality of `MLPClassifierOwn` to handle binary classification tasks, making it more versatile for different types of classification problems.

3.3 Task 2:Problem

The code skeleton will create a binary classification dataset by slicing out the subset of the Sign Language dataset where the label is 0 or 1 (neglecting all other classes). In `nn_classification_from_scratch.py`, use the `train_nn_own` function from the previous task to train the model on this subset of data. After training, use the `score` method of the classifier to compute the accuracy on the train and test set. Report the final train, validation and test accuracy in your report.

3.4 Task 2:Solution

To test the binary classification capabilities, we created a binary classification dataset by slicing out the subset of the Sign Language dataset where the label is 0 or 1 (neglecting all other classes).

In `nn_classification_from_scratch.py`, we used the `train_nn_own` function from the previous task to train the model on this subset of data. After training, we used the `score` method of the classifier to compute the accuracy on the train and test sets.

3.4.1 Final Accuracy

- Train Accuracy: 99.62%
- Validation Accuracy: 98.48%
- Test Accuracy: 96.34%

3.4.2 Interpretation of Results

The reported training results show high accuracy and low loss values, which might suggest that the model is highly effective. However, the following considerations should be taken into account:

- **Overfitting**: The high training accuracy might indicate overfitting, especially if the validation accuracy is lower. Overfitting occurs when the model learns the training data too well, including its noise and outliers, which negatively impacts its performance on unseen data.
- **Regularization**: Introducing regularization techniques such as dropout, L2 regularization, or data augmentation might help in reducing overfitting and improving the model's generalization capabilities.
- **Data Imbalance**: If the dataset is imbalanced, the model might achieve high accuracy by being biased towards the majority class. In such cases, other metrics like precision, recall, and F1-score should be considered to get a better understanding of the model's performance.

3.5 Task 3:Problem

(Theory Question) Your friend comes to you and says: "I have changed the labels in the Sign Language dataset in the following way: I leave every image with class 0 untouched, but I change the label of every image with a class id $\in \{1, 2, \ldots, 9\}$ to class 1. Using this new dataset with binary labels, I have trained a binary classifier, which achieves a test accuracy of $\approx 90\%$. That's great, don't you think?" Is accuracy a misleading metric in this case? Explain why/why not. If you think that it is misleading, explain which performance metric is more useful in this case.

3.6 Task 3:Solution

Accuracy can indeed be a misleading metric in this case. Here's a detailed explanation of why this is the case and which performance metric would be more useful:

Imbalanced Dataset

- The dataset has been transformed into a binary classification problem where class 0 remains as is, but classes 1 through 9 are merged into class 1.
- If the dataset is significantly imbalanced (e.g., there are many more images of class 0 than class 1), a high accuracy can be achieved by simply predicting the majority class for most instances.

Impact of Imbalance

- Suppose 90% of the dataset consists of images from class 0 and only 10% from class 1. A classifier that always predicts class 0 will achieve an accuracy of 90%, which seems impressive, but it doesn't actually perform well for class 1.
- This means that the classifier is not learning to distinguish between the two classes but rather leveraging the imbalance in the dataset to achieve high accuracy.

Why Accuracy is Misleading

- Accuracy measures the proportion of correctly predicted instances over the total instances. In an imbalanced dataset, it can be skewed by the majority class, providing a false sense of model performance.
- In this case, achieving 90% accuracy might simply mean that the classifier is predicting class 0 most of the time, regardless of the actual class.

More Useful Performance Metrics

Precision

- Precision for class 1 is the ratio of true positive predictions to the total predicted positives.
- Precision answers the question: "Of all the instances classified as positive, how many are actually positive?"

Recall (Sensitivity)

- Recall for class 1 is the ratio of true positive predictions to the total actual positives.
- Recall answers the question: "Of all the actual positive instances, how many were correctly classified?"

F1 Score

- The F1 score is the harmonic mean of precision and recall. It provides a single metric that balances both the false positives and false negatives.
- F1 score is especially useful in the case of imbalanced datasets, as it considers both false positives and false negatives.

Confusion Matrix

- A confusion matrix provides a detailed breakdown of true positives, true negatives, false positives, and false negatives.
- It helps in understanding the types of errors the classifier is making and gives a clearer picture of its performance.

Conclusion

In summary, accuracy is a misleading metric in this context because it doesn't account for the imbalance in the dataset. Metrics like precision, recall, and F1 score, along with a confusion matrix, provide a more comprehensive and accurate evaluation of the model's performance in distinguishing between the two classes. These metrics ensure that both classes are appropriately considered, and the model's ability to correctly identify the minority class is evaluated.

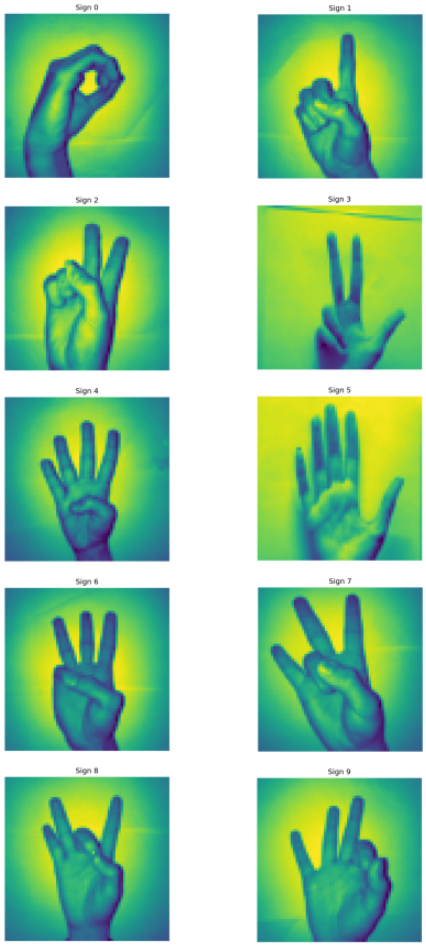

Figure 3: Example image for each class of the Sign Language dataset.