Project B, CS 135
deekshithkomira
Fall - 2023
Tufts University

Problem 1: MLP Classifier

1A: Dataset Exploration

Table 1A with caption

Class Name	Train Set	Validation Set
dress	400	100
pullover	100	100
top	1	100
trouser	1	100
sandal	800	100
sneaker	800	100

Table 1A shows the distribution of the training and validation datasets across the six classes that we have. The major challenge here is that the **data is imbalanced** since, for some of the classes, there are not enough training samples. In the training set, for **the class 'top' and 'trouser,' there is only one training sample for each**. With this imbalanced training data, any **machine learning model will find it difficult to learn patterns for these two classes**, making the model under-perform for these two classes. **The same can be applicable to the class 'pullover'** since it has 100 samples, which could be less in the overall context since there are 400 training samples for the class 'dress' and the classes' sandal' and 'sneaker' have 800 training samples each.

1B: Model Development (including training and hyperparameter selection)

Preprocessing:
I converted the training and the validation arrays to **float32** and **divided the values by 255.0** (the maximum pixel value possible). **Converting to float32** helps since it helps in **numerical stability, compatibility with frameworks, and efficient processing** during training. Dividing by the maximum pixel value helps since it will bring all the values between 0 and 1. **Neural networks are sensitive to the scale of the data, and this process of normalization helps**. When input features are on different scales, appropriate initializations of weights are challenging, and when the gradients are on different scales, it can lead to slow convergence or even prevent the model from converging altogether.

Performance metric for hyperparameter search:
I use **balanced accuracy** as the performance metric during the hyperparameter search for the simple reason that the **final model will be evaluated based on this metric on the test data**. The other reason is that since we have an imbalanced training dataset, **balanced accuracy gives an equal weight to each class, making it more suitable**. It provides a better indication of how well a model is performing across all classes, preventing a skewed evaluation due to the dominance of one or more classes.

MLP's architecture:
For the **only hidden layer**, I used a rule of thumb – the number of hidden units is usually 2/3*(number of input units + number of output units). Using this, I picked **512 (~ 2/3*(784+6)) as the number of units** for the one hidden layer. For this hidden layer, I tried two activation functions – relu (most commonly used) and identity. Surprisingly, identity worked better than relu, and hence, I used **identity as the activation function**.

For the **output layer**, I used **6 units** with **softmax** as the **activation function** since we are building a **6-class classifier.** For this identified MLP architecture, I included an **L2 regularization** (denoted by alpha in sklearn's implementation). I did a comprehensive search for the value of alpha as described in the caption for Table 1B.

MLP's optimization:

I picked **adam as the solver** since I wanted to use the option of early stopping. Sklearn's implementation for early stopping only supports adam and sgd solvers and not the lbfgs solver. Sklearn's documentation also mentions that adam performs pretty well on relatively large datasets (with thousands of training samples or more). **I wanted to use early stopping since, by that, I can implicitly control the number of iterations and not explicitly tune that hyperparameter.** For the adam solver, the other two main hyperparameters are learning rate and batch size. I didn't perform a comprehensive search for them, but I tried the default values and 1-2 values below and above the default values. I monitored the general shift in the validation scores for the range of alpha values and picked the best values. The **best approximately** determined values for all the alphas were – **0.1 for learning rate and 200 for batch size**. The random state is the other hyperparameter for which I did a comprehensive search for each alpha value. More details about this are in the caption of Table 1B.

Table 1B with caption

Alpha Value	Training Balanced Accuracy	Validation Balanced Accuracy
0.0001	0.970	0.778
0.001	**0.991**	**0.870**
0.01	0.986	0.847
1	0.792	0.743
10	0.613	0.618
100	0.557	0.557

I did a comprehensive search for the value of alpha (L2 regularization). **Higher values of alphas equate to more regularization and help in preventing overfitting and obtaining a generalized model**. The range of values shown in Table 1B is log-spaced from 10^-4 to 10^2. Given the wide range of values, I anticipated that at alpha = 10^-4, there would be overfitting, and at alpha = 10^2, there would be underfitting, and I should be able to find the sweet spot in between. **For each value of alpha, I tried 10 (0 – 10) values for the random state, picked the best-performing model on the validation set, and reported it in the table**. This is done to account for the randomness associated with the weight initializations. The key takeaways from this hyperparameter search are:

(i) The training balanced accuracy has an approximate decreasing trend as we increase the alpha value. This is expected since as the alpha value increases, the weights tend to 0, making the model underfit on the training data.

(ii) The validation balanced accuracy first increases, indicating the model is going from the overfitting regime to a good generalized model. Thereafter, the validation accuracy decreases, indicating that it is entering the underfitting regime.

(iii) Overall, at **alpha = 0.001**, we get the **best validation balanced accuracy** of **0.870**. Hence, this alpha value is picked as the optimal value.

1C: Model Analysis
Table 1C with caption

Balanced Accuracy on the Validation Set: 0.870

		Predicted Class					
		dress	**pullover**	**top**	**trouser**	**sandal**	**sneaker**
True Class	**dress**	96	2	0	1	1	0
	pullover	2	98	0	0	0	0
	top	11	18	68	1	2	0
	trouser	24	6	0	70	0	0
	sandal	0	0	0	0	93	7
	sneaker	0	0	0	0	3	97

The model performs **really well (>90% accuracy) for the following classes: dress, pullover, sandal, and sneaker**. When it comes to the **classes top and trouser**, it performs reasonably; **only ~70% of samples are correctly classified**. This **limitation** stems from the fact that the **training data had only one sample for each of these classes**. Since there was very limited data available during the training phase for these classes, the model was not able to learn patterns for them and performed badly on the validation set. The other observation here is that the misclassified samples from the top and trouser classes belong to some form of clothing category and not to the footwear category. This made sense since the images of footwear are quite different from the images of clothing.

1D: Training Set Modification
I followed the below steps to duplicate the data:

(i) Separated the data for the classes for which I wanted to duplicate the samples.
(ii) Made the required number of copies of the samples.
(iii) Joined all the duplicated data with the original data for other classes that were not duplicated and shuffled all the data. I believe shuffling is important since I used adam optimizer, which uses the concept of batches, and I didn't want to have some of the batches with just data from one class.

The table below describes the **final composition of the training and validation sets** after duplicating the data that is fed into the modeling process explained in IE.

Class Name	Train Set	Validation Set
dress	400	100
pullover	400	100
top	400	100
trouser	400	100
sandal	800	100
sneaker	800	100

To decide the final composition of the classes after duplication, I considered two things – **make each class representative** so that the model performs well on each class separately and also make sure that the **model is not overfitting** on the training data since we are just duplicating the data, making it more prone to overfitting. **By duplicating the data for the under-representative classes, I expect the model to perform better for those classes and, on an overall level, get an improved validation balanced accuracy.** I didn't try just one strategy but tried three strategies, as described below, to determine the best strategy.

To start with, I increased the count of samples belonging to the class top and trouser to 100 and reran the modeling process. I observed an increase in the validation's balanced accuracy. To increase the accuracy further, I increased the count of samples to 400 for the top and trouser classes. Also, I increased the count of samples to 400 for the pullover class. By doing this, I had an equal number of samples for all the clothing classes. And, the footwear classes also have the same number of classes. I saw a further increase in the validation balanced accuracy. Going further, by increasing the number of samples for the clothing classes to 800 to make it equal to footwear classes, I didn't observe an increase in the validation's balanced accuracy. This made sense since the images of footwear are quite different from the images of clothing. Hence, out of these three strategies, the second strategy work the best, and hence, I settled for that.

1E: Duplicated-Data Model Development

I followed the same preprocessing technique, the same performance metric for hyperparameter search, and the same MLP architecture mentioned in 1B. I used the adam optimizer with early stopping for the reasons mentioned in 1B. Doing a similar simplified search for learning rate and batch size as mentioned in 1B, I determined the **approximate best values** for all the alphas to be **0.1 for learning rate and 200 for batch size**. I did a comprehensive search for the hyperparameters – alpha and a random state value. This is further explained in the caption of Table 1E.

Table 1E with caption

Alpha Value	Training Balanced Accuracy	Validation Balanced Accuracy
0.0001	0.992	0.907
0.001	0.987	0.913
0.01	0.993	0.897
1	**0.966**	**0.920**
10	0.948	0.883
100	0.778	0.722

The range of values shown in Table 1E is log-spaced from 10^-4 to 10^2. **For each value of alpha, I tried 15 (0 – 15) values for the random state and picked the best-performing model on the validation set, as reported in the table**. The key takeaways from this hyperparameter search are:

(i) The training balanced accuracy has an approximate decreasing trend as we increase the alpha value. This is expected since as the alpha value increases, the weights tend to 0, making the model underfit on the training data.

(ii) The validation balanced accuracy first increases, indicating the model is going from the overfitting regime to a good generalized model. Thereafter, the validation accuracy decreases, indicating that it is entering the underfitting regime.

Overall, at **alpha = 1**, we get the **best validation balanced accuracy** of **0.920**. Hence, this alpha value is picked as the optimal value.

1F: Modified Model Analysis

Table 1F with caption

Balanced Accuracy on the Validation Set: 0.920

		Predicted Class					
		dress	**pullover**	**top**	**trouser**	**sandal**	**sneaker**
True Class	**dress**	95	2	2	1	0	0
	pullover	2	97	1	0	0	0
	top	5	11	82	0	2	0
	trouser	8	8	1	83	0	0
	sandal	0	2	0	0	95	3
	sneaker	0	0	0	0	0	100

One sharp contrast of this result when compared to the results of 1C is that the **accuracy for the top and the trouser classes increased to ~82% from ~70% obtained in 1C**. This was achieved without compromising the accuracy for the other classes; the other classes still have >90% accuracy. This was something I expected based on the modification procedure I used. **By increasing the number of samples for the top and trouser classes, these classes were made representative,** and the machine learning model was able to learn patterns for these classes and perform way better. **But it still didn't quite touch the accuracy of other classes since the new data added lacks variety and is essentially the same data duplicated.**

1G: Submit to Leaderboard and Record Test-Set Performance

The balanced accuracy on the test set obtained is **0.886**. This value is close to the validation's balanced accuracy (0.920). Since the test data is unseen, very similar values establish that our model did not overfit, but the validation accuracy can be better estimated by using robust cross-validation techniques.

Problem 2: Open-Ended Challenge

2A: Method Description

Data Augmentation / Preprocessing:

I take the dataset developed in 1D to start with. Using this dataset, I **combine both the training and the validation sets** and get combined data. Then**, I split this dataset using the sklearn's train_test_split function, with 30% for validation and 70% for training.** I hypothesize that this will help to improve the model's performance since all the duplicated data is not just present in the training data; instead, it is distributed across the training and the validation sets. This will **bring variety to the training data** and, hence, will help improve performance. The composition of the training and validation sets is shown in the below table.

Class Name	Train Set	Validation Set
dress	352	148
pullover	347	153
top	357	143
trouser	362	138
sandal	606	294
sneaker	636	264

Feature Representation:

Not investigated

Classification Method:

I use **Convolutional Neural Networks (CNNs)** as the classification method. The main motivation for choosing this classification method when compared to any other classification algorithms is that **CNNs have proved to be superior in the domain of image recognition and classification tasks since their architecture is specifically designed to capture spatial hierarchies and local patterns in visual data**. CNNs leverage convolutional layers that employ filters to detect local features like edges, textures, and shapes across different regions of an image. Additionally, CNNs often employ pooling layers to downsample and preserve essential information while reducing the computational complexity. These are the reasons why I think this method will help me achieve a better performance for this dataset of images.

To implement CNN in Python, I use **tensorflow.keras** module. This module provides a sub-module called models, which has a function called **Sequential**. This function allows the addition of **convolution, pooling, dense, and output layers sequentially and, therefore, allows the development of complex models**. The module also offers regularizers and callbacks to implement **L2 regularization and early stopping**. This module expects the data to be in a different structure; the preprocessing applied for that is explained below:

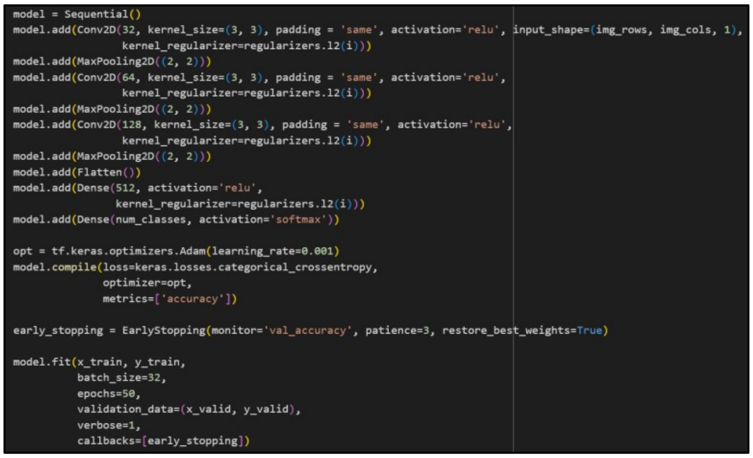
(i) The **class labels** are converted to one-hot encoding array formats. This can be achieved by the **keras.utils.to_categorical()** function.

(ii) The pixel data for gray-scaled images is expected to be in the format of (rows, cols, 1). The dataset was given to us in the format of 784 columns for each image. This is **reshaped into (28, 28, 1) arrays** for each image. Even here, the pixel values were converted into float32 values and divided by 255.0 (the maximum pixel value).

The model's architecture, hyperparameter tuning, and model optimization details are mentioned in the 2B section.

2B: Model Development (including training and hyperparameter search)

To develop the architecture, I took inspiration from **Figure 10.8 on Page No. 411 in the ISLP book**. The below image gives details about the architecture and the optimization of the model.



```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), padding = 'same', activation='relu', input_shape=(img_rows, img_cols, 1),
                 kernel_regularizer=regularizers.l2(i)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), padding = 'same', activation='relu',
                 kernel_regularizer=regularizers.l2(i)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(128, kernel_size=(3, 3), padding = 'same', activation='relu',
                 kernel_regularizer=regularizers.l2(i)))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(512, activation='relu',
                kernel_regularizer=regularizers.l2(i)))
model.add(Dense(num_classes, activation='softmax'))

opt = tf.keras.optimizers.Adam(learning_rate=0.001)
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=opt,
              metrics=['accuracy'])

early_stopping = EarlyStopping(monitor='val_accuracy', patience=3, restore_best_weights=True)

model.fit(x_train, y_train,
          batch_size=32,
          epochs=50,
          validation_data=(x_valid, y_valid),
          verbose=1,
          callbacks=[early_stopping])
```

I have **three sets of convolution and max pooling layers**. For the convolution and max pooling layers, I use the standard values obtained from the ISLP book for all the hyperparameters. 'Same' padding is used to retain the size of the image after each convolution layer. After passing through these three sets of layers, the size of each image will be in the format of (3, 3, 128). This is flattened into an array of size (1152) and then is passed into **a hidden layer of size 512** (hidden units determined by the rule of thumb mentioned in 1B). Then, finally, it goes into an output layer. Adding additional fully connected hidden layers was investigated, and it didn't improve the balanced accuracy.

As part of model complexity hyperparameters, the value of alpha was searched. **L2 regularizations** were included in all the **convolution layers** and one **hidden layer**. **Since we are using duplicated images, I felt overfitting would be an important issue to be addressed,** and hence, more emphasis was given to it. **Higher values of alphas equate to more regularization and help in preventing overfitting and obtaining a generalized model**. The range of values tried was **10^-5 to 1**, a total of six values, as mentioned in the Table 2B. Given the wide range of values, I anticipated that at alpha = 10^-5, there would be overfitting, and at alpha = 10^0, there would be underfitting, and I should be able to find the sweet spot in between. A **fixed validation set** was used for the hyperparameter search using **balanced accuracy**. Balanced accuracy was used for the same reasons as mentioned in 1B.

As part of optimization quality hyperparameters, a comprehensive search was not performed. Instead, a simplified search was performed. I started with the default values, and 1-2 values below and above the default values were investigated. I monitored the general shift in the validation scores for the range of alpha values and picked the best values. The **best approximately** determined values for all the alphas were **0.001** for the learning rate and **32** for the **batch size**. For the **epoch size**, a very big number of **50 was set,** and then **early stopping was included with a**

patience value of 3. When the validation accuracy does not increase for three epochs, the model training is stopped, and the last best weight values are restored. A small caveat here is that this is the regular accuracy and not the balanced accuracy. Implementing balanced accuracy for this purpose was complicated and, hence, was not done.

Table 2B with caption:

Alpha Value	Training Balanced Accuracy	Validation Balanced Accuracy
0.00001	0.994	0.981
0.0001	**0.995**	**0.982**
0.001	0.984	0.974
0.01	0.955	0.952
0.1	0.881	0.879
1	0.167	0.167

The range of values shown in Table 2B is log-spaced from 10^-5 to 10^0. The key takeaways from this hyperparameter search are:

(i) The training balanced accuracy has an approximate decreasing trend as we increase the alpha value. This is expected since as the alpha value increases, the weights tend to 0, making the model underfit on the training data.

(ii) The validation balanced accuracy first increases, indicating the model is going from the overfitting regime to a good generalized model. Thereafter, the validation accuracy decreases, indicating that it is entering the underfitting regime.

Overall, at **alpha = 0.0001**, we get the **best validation balanced accuracy** of **0.982**. Hence, this alpha value is picked as the optimal value. Mixing the training data and validation data brings variety to the training data, and it is able to give a bump to the validation score. Further, using CNN algorithms, which are specifically designed for images, gives a further bump to the validation score.

2C: Model Analysis

Table 2C with caption

Balanced Accuracy on the Validation Set: 0.982

		Predicted Class					
		dress	**pullover**	**top**	**trouser**	**sandal**	**sneaker**
True Class	**dress**	144	2	1	1	0	0
	pullover	1	152	0	0	0	0
	top	0	3	139	0	1	0
	trouser	0	1	0	137	0	0
	sandal	0	0	0	0	285	9
	sneaker	0	0	0	0	3	261

In the case of **1C**, the model was performing badly **(~70%)** for the top and trouser classes since there was just one sample for each class. This was improved in **1F** to **~82%** by adding duplicated data for these classes. This is further improved significantly **(~98%)** by **changing the composition of the training data and using a CNN classifier**. This was achieved without compromising the accuracy for the other classes; the other classes still have >90% accuracy. The results support the first hypothesis made in 2A since I was able to bring in an improvement of ~16% when compared to 1F for the two classes for which a lot of samples were duplicated. The second hypothesis is also supported by the results since we see an overall improvement across all the classes when compared to the results obtained in 1F.

2D: Submit to Leaderboard and Record Test-Set Performance

The balanced accuracy on the test set obtained is **0.936**. This value is a bit worse when compared to the validation's balanced accuracy (0.982). Comprehensive hyperparameter tuning and implementing robust cross-validation techniques will help us to determine the hyperparameters that don't overfit the validation data and get a better estimate of the validation accuracy.