Deep Learning

DINH HOANG VIET PHUONG – 301123263

LAB 02

Analysis report

**Import all necessary libraries**

A screen shot of a computer program

Description automatically generated

numpy and pandas for data manipulation and analysis.

seaborn and matplotlib.pyplot for data visualization.

tensorflow, specifically the Keras API, to build and train machine learning models.

sklearn.metrics for evaluating model performance, specifically using a confusion matrix.

sklearn.model\_selection for splitting datasets into training and testing sets.

**A. Get the data**

A computer screen shot of a computer program

Description automatically generated

The code loads the Fashion MNIST dataset using TensorFlow's Keras API. Fashion MNIST is a dataset of Zalando's article images, consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes.

The dataset is split into training (train\_images, train\_labels) and testing (test\_images, test\_labels) sets.

Two dictionaries, unsupervised\_phuong and supervised\_phuong, are created to organize the data for different types of machine learning tasks:

* unsupervised\_phuong is prepared with only the images from the training set, indicating it could be used for unsupervised learning tasks where labels are not required.
* supervised\_phuong includes both images and labels from the test set, indicating it's prepared for supervised learning tasks where the model learns from labeled data.

**B. Data Pre-preprocessing**

A screen shot of a computer program

Description automatically generated

Normalizing Pixel Values:

* Normalization is applied to both the unsupervised\_phuong['images'] and supervised\_phuong['images'] datasets. This process converts the pixel values from integers in the range [0, 255] to floating-point numbers in the range [0, 1].
* This is achieved by casting the image arrays to float32 data type and dividing by 255. Normalization is essential for deep learning models as it makes the training process more stable and faster by ensuring all input features (pixel values in this case) are on a similar scale.

One-hot Encoding Labels:

* One-hot encoding transforms the labels in supervised\_phuong['labels'] into a binary matrix representation. This is necessary for classification tasks where each label is converted into a binary vector of length equal to the number of classes (10 for Fashion MNIST), with a 1 in the position of the class it belongs to and 0s elsewhere.
* This encoding is performed using TensorFlow's tf.keras.utils.to\_categorical function, specifying 10 classes. One-hot encoding is critical for multi-class classification as it allows the model to treat the problem as a single label for each class prediction, simplifying the output layer design and the calculation of loss during training.

**C. Data Preparation (Training, Validation, Testing)**

A computer screen with text on it

Description automatically generated

1. Splitting the Unsupervised Dataset

* The unsupervised dataset's images are flattened from a 2D (28x28) format to a 1D (784) format. This step is necessary for models that require a flat input vector rather than a 2D array.
* Train-test split is performed on the flattened unsupervised dataset to create training and validation sets, using 57,000 images for training and 3,000 for validation. The split uses a dummy label array since labels are not needed for unsupervised learning tasks.
* The resulting training and validation datasets are stored in Pandas dataframes, facilitating easier data manipulation and analysis.

A computer screen shot of a program code

Description automatically generated

2. Discarding Samples from the Supervised Dataset

* The supervised dataset undergoes a similar flattening process. Then, a subset of the data is discarded to keep only 3,000 samples, simulating a scenario where the dataset size is intentionally reduced for experimentation or due to resource constraints.
* This is achieved by performing a train-test split and keeping the smaller portion as the new dataset.
* The supervised\_phuong dictionary is updated to reflect this reduced dataset.

A screen shot of a computer program

Description automatically generated

3. Splitting the Remaining Supervised Dataset

* The remaining supervised dataset is then split into training, validation, and testing sets. The dataset is first split to separate out 600 samples for testing. The rest is further split to allocate 600 samples for validation out of the remaining 2,400, leaving 1,800 for training.
* This structured split ensures a clear separation of data for training models, validating their performance during tuning, and finally testing them to evaluate their generalization capability.
* The datasets are stored in Pandas dataframes for ease of use, while the labels, already in a one-hot encoded format, are directly assigned to corresponding variables.

A screenshot of a computer program

Description automatically generated

4. Confirmation of Dataset Sizes

* The shapes of all prepared datasets are printed to confirm their sizes. The training, validation, and test splits for both unsupervised and supervised learning tasks are now clearly defined and ready for use:
* Unsupervised learning datasets have 57,000 training samples and 3,000 validation samples, all flattened to 784 features.
* Supervised learning datasets have been reduced and split into 1,800 training, 600 validation, and 600 testing samples, with each image flattened to 784 features. The labels for these samples are in a one-hot encoded format, indicating membership across 10 classes.

**D. Build, Train, and Validate a baseline CNN Model**

A computer screen shot of a program

Description automatically generated

1. The model, cnn\_v1\_model\_phuong, is defined using TensorFlow's Keras API with a sequential model architecture. This architecture is specifically designed for image classification tasks and includes the following layers:

* An Input Layer specifying the shape of the input data (28x28 images with 1 color channel, indicating grayscale images).
* Two Conv2D layers with ReLU activation functions. The first has 16 filters and the second has 8 filters, both with a kernel size of 3x3, using 'same' padding and a stride of 2. These layers are responsible for extracting features from the input images through convolution operations.
* A Flatten layer to convert the 2D feature maps into a 1D vector, making it suitable for input into the dense layers that follow.
* A Dense layer with 100 neurons and ReLU activation, serving as a fully connected layer that interprets the features extracted by the convolutional layers.
* A final Dense layer with 10 neurons (corresponding to the 10 classes of the Fashion MNIST dataset) and softmax activation to output probabilities for each class.

2. Compiling the Model: The model is compiled with the Adam optimizer, categorical crossentropy as the loss function (appropriate for multi-class classification), and accuracy as the metric for performance evaluation.

A screenshot of a computer program

Description automatically generated

3. Model Summary and Visualization: The summary of the model displays the architecture, showing each layer's type, output shape, and the number of parameters. The total trainable parameters are 41,630, indicating the complexity and capacity of the model to learn from the data.

4. Preparing the Input Data

* The input data (x\_train\_phuong and x\_val\_phuong) is reshaped to include the channel dimension, making it compatible with the CNN model's expected input shape. This involves reshaping the data from a 2D format (28x28) to a 3D format (28x28x1).
* Training the Model: The model is trained using the reshaped training data and the one-hot encoded labels (y\_train\_phuong). The validation dataset (x\_val\_reshaped and y\_val\_phuong) is used to monitor the model's performance on unseen data during training. The training process is carried out for 10 epochs with a batch size of 256, balancing the trade-off between training speed and memory usage.

**E. Test and analyze the baseline model**

A screen shot of a computer program

Description automatically generated

A graph with a line and a red and blue line

Description automatically generated

1. Plot Training vs Validation Accuracy

The provided plot illustrates the training and validation accuracy of the baseline CNN model over 10 epochs of training:

* Training Accuracy: It starts at around 30% and increases significantly to just over 70% by the end of epoch 10. The steady increase indicates that the model is learning from the training data.
* Validation Accuracy: It also starts at around 30% and follows a similar upward trend as the training accuracy, reaching slightly below 70% by epoch 10. This parallel increase suggests that the model is generalizing well and not overfitting significantly to the training data.
* Gap Between Training and Validation Accuracy: The gap between the training and validation accuracy remains relatively narrow throughout the epochs, which is a positive sign. It means that the model is not just memorizing the training data but also performing well on data it hasn't seen before.
* Convergence: Both accuracies seem to be plateauing towards the end of the training process, indicating that the model might not significantly improve with more epochs given the current architecture and hyperparameters.

A computer screen shot of a program

Description automatically generated

2. Evaluate the Model on Test Dataset

The model's test accuracy is approximately 76%, as indicated by the evaluation on the test dataset. Here's the comparison and discussion with respect to the validation accuracy:

* Test vs. Validation Accuracy: The test accuracy is higher than the validation accuracy observed during training (which was just below 70%). This is a good indication that the model has generalized well and is performing consistently on unseen data.
* Interpretation: A higher test accuracy compared to validation accuracy can sometimes occur, especially if the test set happens to be slightly easier for the model to predict, or if the validation set during training was more challenging or not fully representative of the overall data distribution.
* Overfitting Check: The fact that the test accuracy is in line with the validation accuracy suggests that there is no significant overfitting. If the test accuracy had been much lower than the validation accuracy, it would indicate that the model might have overfitted to the validation set during hyperparameter tuning or model selection.

A screenshot of a graph

Description automatically generated

Analysis of the Confusion Matrix:

* Diagonal Elements: The values along the diagonal of the matrix represent the number of correct predictions for each class. The darker the square, the higher the number of correct predictions. For example, we can see that the model correctly predicted 34 instances of class '0'.
* Off-diagonal Elements: These elements represent instances where the model predicted incorrectly. For instance, the model seems to confuse class '0' with class '8', as indicated by the value '8' in the off-diagonal position (row 0, column 8).
* Class Performance: Some classes seem to be predicted with higher accuracy than others. Class '0', despite some misclassification, appears to have a higher number of correct predictions. However, the actual number of samples in each class isn't shown, so it's not possible to determine the per-class accuracy from this matrix alone.
* Misclassifications: There are visible misclassifications, such as between classes '2' and '6', '3' and '5', etc. This might be indicative of visual similarities between these classes that the model is struggling to distinguish.

Discussion:

* Model Strengths: The presence of higher values along the diagonal suggests that the model has learned to correctly classify a significant number of instances across different classes.
* Model Weaknesses: The off-diagonal numbers indicate where the model is most frequently making mistakes. For example, if classes '2' and '4' have a high number of misclassifications, it could be that the features the model is learning are not distinctive enough between these classes, or the model architecture is not complex enough to capture the difference.
* Improvement Areas: To improve the model's performance, we could look into data augmentation techniques to make the model more robust to variations within each class. Additionally, increasing model complexity, tuning hyperparameters, or even gathering more data could potentially improve the model's ability to distinguish between the more frequently confused classes.
* Class Imbalance: If there's a class imbalance in the dataset, the model might be biased towards predicting classes with more samples. It's important to look at the distribution of classes in the dataset and consider techniques like class weighting or resampling to mitigate this.
* Error Analysis: A detailed error analysis could reveal specific patterns in the misclassifications that could inform improvements in data preprocessing, feature engineering, or model architecture.

**F. Add random noise to unsupervised dataset**

A screenshot of a computer

Description automatically generated

1. Add Random Noise

* This step involves adding random noise to the unsupervised training and validation datasets to simulate more challenging real-world conditions and possibly improve the robustness of models trained on this data.
* Noise Addition: A noise factor of 0.2 is used to scale the random noise generated by TensorFlow's tf.random.normal function, which produces a tensor of random values following a normal distribution. This noise is added to the unsupervised datasets (unsupervised\_train\_phuong and unsupervised\_val\_phuong), initially converted to numpy arrays and reshaped to include the channel dimension.
* Value Clipping: After adding the noise, the pixel values are clipped to the range [0, 1] to ensure they remain valid grayscale values. This step is necessary because the addition of noise can lead to values outside this range.

2. Clip Values: Although the datasets have already been clipped to ensure pixel values are within the [0, 1] range, this step appears to redundantly apply clipping again using TensorFlow's tf.clip\_by\_value. This operation is applied to the pandas DataFrames converted from the noisy numpy arrays, ensuring values are within the desired range. However, since the clipping was already performed on the numpy arrays, this step may not be necessary if the data conversion back to DataFrames retains the clipped values correctly.

3. Plot the Noisy Images

* Visualization: A selection of noisy images from the validation dataset is visualized to illustrate the effect of the added noise. This visualization helps in understanding the impact of noise on the data and potentially on model performance.
* Implementation Details: The plotting is done using Matplotlib, displaying 10 images in a row to provide a clear view of the noise effect. Each image is shown in grayscale to reflect the original data format.

**G. Build and pretrain Autoencoder**

A computer screen shot of a program code

Description automatically generated

A screenshot of a computer program

Description automatically generated

1. Building the Autoencoder

The autoencoder is designed using TensorFlow's functional API, consisting of an encoder and a decoder section:

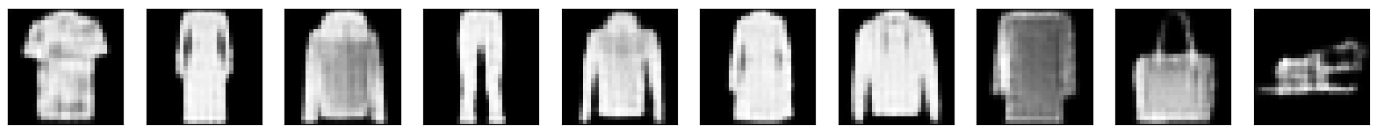
* Encoder: Compresses the input image into a lower-dimensional latent space using convolutional layers. The encoder has two convolutional layers with 16 and 8 filters, respectively, both applying 'relu' activation, 'same' padding, and a stride of 2 to downsample the input.
* Decoder: Attempts to reconstruct the input image from the latent representation. The decoder uses two transposed convolutional layers (also known as deconvolutional layers) with 8 and 16 filters, respectively, followed by a final convolutional layer that outputs the reconstructed image. The activation function for the final layer is 'sigmoid', ensuring the output pixel values are in the [0, 1] range.

2. Compiling the Model: The autoencoder is compiled with the 'adam' optimizer and 'mean\_squared\_error' loss function. This setup is typical for reconstruction tasks where the goal is to minimize the difference between the original and reconstructed images.

3. Model Summary and Visualization: The model summary reveals the architecture and parameters of the autoencoder. Additionally, a graphical representation of the model is generated, illustrating the flow from input to reconstructed output, facilitating an understanding of the model's structure.

A screenshot of a computer program

Description automatically generated



4. Training and Validation: The autoencoder is trained on noisy images (as input) and original images (as target output) for 10 epochs with a batch size of 256. This training process teaches the autoencoder to denoise images, learning to map noisy inputs to clean outputs. Validation is performed on a separate set of noisy and original images to monitor the model's performance on data not seen during training.

5. Creating Predictions: After training, the autoencoder is used to predict on the validation set, generating denoised versions of the input images. These predictions demonstrate the model's ability to reconstruct cleaner images from the noisy inputs.

6. Displaying Predicted Images: The first 10 predicted images are displayed, showcasing the results of the denoising process. This visualization helps assess the effectiveness of the autoencoder in removing noise and reconstructing images that resemble the original, clean versions.

**H. Build and perform transfer learning on a CNN with the Autoencoder**

A computer screen shot of a program

Description automatically generated

1. Building the CNN Model with Transferred Encoder Layers

This approach utilizes the encoder portion of the pre-trained autoencoder as the initial layers of a new Convolutional Neural Network (CNN) model for classification:

* Transferred Encoder Layers: The encoder layers from the autoencoder are reused, leveraging their ability to extract and compress relevant features from the input images.
* Flatten Layer: The output of the encoder is flattened to transform the data into a format suitable for fully connected layers.
* Fully Connected Layer: A new dense layer with 100 neurons and 'relu' activation is added to interpret the features extracted by the encoder.
* Output Layer: The model concludes with a dense layer with 10 units and 'softmax' activation, designed for multi-class classification (assuming there are 10 classes).

2. Compiling the Model

The new CNN model is compiled with:

* Optimizer: 'adam', a popular choice for training deep learning models due to its efficiency.
* Loss Function: 'categorical\_crossentropy', appropriate for multi-class classification tasks where labels are one-hot encoded.
* Metrics: 'accuracy', to monitor the model's performance.

3. Model Summary and Visualization: The model summary reveals the architecture, showing the flow from the encoder input through to the classification output. A graphical representation of the model is also generated, illustrating its structure, which is helpful for understanding how the transferred layers integrate with the new classification layers.

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

4. Training and Validation: The new CNN model is trained on reshaped training data (x\_train\_phuong\_reshaped) with labels (y\_train\_phuong), and validated on reshaped validation data (x\_val\_phuong\_reshaped) with labels (y\_val\_phuong). This process fine-tunes the transferred encoder layers to the specific classification task at hand, optimizing the model's performance on the target dataset.

**I. Test and analyze the pretrained CNN model**

A computer screen shot of text

Description automatically generated

1. Plot Training vs Validation Accuracy

The plot provided shows the training and validation accuracy of the pretrained CNN model over 10 epochs. Observations from the plot include:

* Training Accuracy: The training accuracy starts at approximately 50% and steadily increases to just over 70% by the 10th epoch.
* Validation Accuracy: The validation accuracy begins slightly lower than the training accuracy but quickly surpasses it, indicating good generalization. However, it plateaus and slightly fluctuates after the 6th epoch, suggesting that the model may not be significantly benefiting from additional training past this point.
* Overfitting: There is no significant overfitting observed as the validation accuracy follows the training accuracy closely, without a substantial divergence.

A graph of a graph

Description automatically generated

2. Evaluate the Model on the Test Dataset

The model achieved a test accuracy of approximately 79.17%, which is a good sign of the model's generalization capability.

Comparing Training, Validation, and Test Accuracy

* Training vs. Validation Accuracy: The close performance of training and validation accuracy suggests that the model has learned patterns that generalize well to unseen data. The slight fluctuations in validation accuracy could indicate minor overfitting or simply represent the variability in the validation dataset.
* Validation vs. Test Accuracy: The test accuracy (79.17%) is higher than the final reported validation accuracy from the training epochs, which is a positive outcome, as it suggests that the model performs well on completely unseen data and not just the data used for tuning during the validation phase.
* General Observations: The model seems to provide a stable performance across training, validation, and testing phases, which is indicative of its robustness. The fact that the test accuracy is higher than the training accuracy may also suggest that the test set may have been slightly easier for the model to predict, or that the model has successfully captured the underlying patterns without overfitting.

Discussion

* The pretrained CNN model exhibits a strong generalization performance, as evidenced by the high-test accuracy in comparison to the validation accuracy. This implies that the features learned during the pretraining with the autoencoder are effective for the classification task. However, it's important to note that to fully validate the model's performance, further testing with additional datasets or using cross-validation methods could be beneficial to ensure consistency of the model's accuracy. Additionally, the plateau in the validation accuracy could be a signal to employ techniques such as early stopping or to explore more complex models or training procedures to push the performance further.

A computer screen shot of text

Description automatically generated

A graph with blue squares

Description automatically generated

Observations from the Confusion Matrix:

* Diagonal Values: The diagonal values represent the number of correct predictions for each class. High values on the diagonal, such as for class 0 and class 9, indicate that the model performs well for these classes.
* Off-Diagonal Values: The off-diagonal values indicate misclassifications. For instance, class 0 was sometimes confused with class 3 and class 6, as indicated by the non-zero off-diagonal values.
* Class Distribution: The distribution of predictions for each class is not uniform. Some classes seem to have a higher number of samples (darker squares), while others have fewer (lighter squares).
* Problematic Classes: Certain classes, such as class 6, show more significant confusion, both in being misclassified as other classes and in having other classes misclassified as class 6.
* Potential Issues: The presence of misclassifications suggests that some classes may be similar to each other, or the features learned by the model are not distinct enough to differentiate between them accurately.

Discussion:

* Model Strengths: The model has a strong predictive ability for some classes, likely due to distinct features that it can reliably detect. For example, class 0 and class 9 have the highest correct predictions, which suggests that the model has learned robust features for these classes.
* Model Weaknesses: The misclassifications indicate areas where the model may be improved. For instance, the confusion between class 0 and class 6 could be due to similarities in the shape or texture patterns that the model is unable to distinguish.
* Improvement Strategies: To improve the model, one could consider additional training data, data augmentation techniques to enhance feature learning, or model architecture adjustments to capture more complex patterns.
* Class Imbalance: If the dataset has a class imbalance, it could lead to a bias in the model's predictions. Strategies such as class weighting during training or oversampling minority classes could address this issue.

**J. Compare the performance of the baseline CNN model to the pretrained model**

A screen shot of a computer program

Description automatically generated

A graph showing the difference between model validation and model validation

Description automatically generated

1. Validation Accuracy Comparison

The provided graph compares the validation accuracy of the baseline CNN model with the pretrained CNN model across 10 epochs of training:

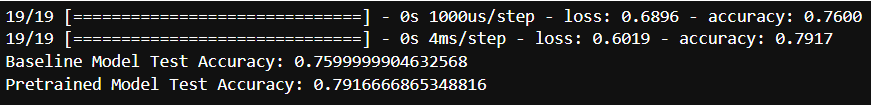
* Baseline Model: The validation accuracy of the baseline model shows a rapid increase in the initial epochs, then starts to plateau around epoch 6, with a slight decrease and fluctuation in the later epochs.
* Pretrained Model: The pretrained model exhibits a generally higher validation accuracy compared to the baseline model. It also shows a rapid increase initially, surpasses the baseline model by epoch 2, and then maintains a relatively steady performance with a slight dip towards the end.

Analysis of Validation Accuracy:

* Early Performance: Both models improve quickly in the initial epochs, which is common as the models begin to learn from the data.
* Mid to Late Performance: The pretrained model maintains a lead over the baseline model, which indicates that the features learned during the pretraining phase with the autoencoder are beneficial for the classification task.
* Fluctuations and Plateaus: The fluctuations observed in both models' accuracies in later epochs suggest that they might be reaching their performance limits with the given architecture and data. It may also indicate a need for hyperparameter tuning or regularization to stabilize learning.
* Generalization Ability: The consistently higher validation accuracy of the pretrained model suggests better generalization compared to the baseline model.

A screen shot of a computer code

Description automatically generated



2. Test Accuracy Comparison

* Baseline Model Test Accuracy: 76.00%
* Pretrained Model Test Accuracy: 79.17%

Analysis of Test Accuracy:

* Improved Performance: The pretrained model outperforms the baseline model on the test dataset, reinforcing the benefits of transfer learning and the utility of the features learned by the autoencoder.
* Consistency with Validation Trends: The test accuracy trends are consistent with the validation accuracy trends, where the pretrained model exhibits better performance.
* Indication of Robustness: The test accuracies further indicate that the pretrained model is more robust, as it has learned more generalizable features that perform well on unseen data.