DEEP LEARNING

CASE STUDY 2

B UDAY KUMAR

HU21CSEN0100964

Train the stacked auto-encoders for the classification of images.

I) What is an Auto Encoder?

A neural network trained using unsupervised learning

• Trained to copy its input to its output

• Learns an embedding

An autoencoder is a type of artificial neural network used for unsupervised learning of efficient data codings. The aim of an autoencoder is to learn a representation (encoding) for a set of data, typically for dimensionality reduction, feature learning, or data denoising, without supervision.

The network architecture of an autoencoder consists of an encoder and a decoder:

Encoder: The encoder compresses the input data into a lower-dimensional latent space representation. It consists of one or more layers of neurons that transform the input data into a compressed representation, often referred to as the code or latent representation.

Decoder: The decoder takes the compressed representation produced by the encoder and attempts to reconstruct the original input data. Similar to the encoder, it consists of one or more layers of neurons that decode the compressed representation back into the original data space.

Autoencoders are useful for various tasks:

- **Dimensionality Reduction**: By learning a compact representation of the input data, autoencoders can reduce the dimensionality of high-dimensional data, making it easier to visualize and analyze.
- **Feature Learning**: Autoencoders can learn meaningful features from raw data, which can be useful for subsequent supervised learning tasks such as classification or regression.
- **Data Denoising**: Autoencoders can be trained to reconstruct clean data from noisy or corrupted input, effectively denoising the data.
- **Anomaly Detection**: Autoencoders can be trained on normal data and used to reconstruct new data samples. Samples that are reconstructed poorly may be considered anomalies.

II) Introduction to Stacked Autoencoders:

Stacked autoencoders represent a powerful variant of traditional autoencoders, offering increased flexibility and capability in learning hierarchical representations of data. An autoencoder is a type of artificial neural network designed to learn efficient representations of input data, often used for tasks such as dimensionality reduction, feature learning, and data denoising. Stacked autoencoders extend this concept by stacking multiple layers of autoencoders on top of each other, forming a deep neural network architecture.

Hierarchical Representation Learning:

The key advantage of stacked autoencoders lies in their ability to learn hierarchical representations of data. Each layer in the stacked architecture learns to capture increasingly abstract and complex features of the input data. The lower layers typically capture low-level features such as edges, textures, or basic patterns, while the higher layers learn more sophisticated representations that combine these low-level features to form higher-level concepts. This hierarchical learning process allows stacked autoencoders to effectively capture the underlying structure and variations in the data.

Layer-wise Training:

Stacked autoencoders are typically trained in a layer-wise manner. This involves training each layer of the autoencoder one at a time while keeping the parameters of previously trained layers fixed. During training, the input data is passed through the network, and each layer learns to extract relevant features from the data. Once training is complete for all layers, the entire stacked architecture is fine-tuned using back propagation to optimize its performance for a specific task, such as classification or reconstruction.

Feature Extraction and Representation:

After training, stacked autoencoders can be used as powerful feature extractors. The activations of the neurons in the hidden layers serve as compressed representations of the input data, capturing its essential characteristics in a lower-dimensional space. These learned features can then be used as input to downstream machine learning models for tasks such as classification, clustering, or anomaly detection. The hierarchical nature of the learned representations makes stacked autoencoders particularly well-suited for complex data with multiple levels of abstraction.

Applications and Advantages:

Stacked autoencoders have found applications in various domains, including computer vision, natural language processing, and bioinformatics. They offer several advantages over traditional shallow autoencoders:

- **Hierarchical Feature Learning**: Stacked autoencoders can learn hierarchical representations of data, allowing them to capture complex patterns and variations more effectively.
- **Improved Generalization:** The hierarchical nature of stacked autoencoders enables better generalization to unseen data by learning more abstract and invariant features.
- **Dimensionality Reduction**: Stacked autoencoders can be used for dimensionality reduction tasks, where the goal is to represent high-dimensional data in a lower-dimensional space while preserving important information.
- **Unsupervised Learning**: Stacked autoencoders can be trained in an unsupervised manner, making them suitable for scenarios where labeled data is scarce or unavailable.
- In summary, stacked autoencoders are a versatile and powerful deep learning architecture capable of learning hierarchical representations of data. Their ability to capture complex patterns and features in an unsupervised manner makes them valuable tools for various machine learning tasks.

III) Implementation:

DONE BY USING MATLAB

Neural networks with multiple hidden layers can be useful for solving classification problems with complex data, such as images. Each layer can learn features at a different level of abstraction. However, training neural networks with multiple hidden layers can be difficult in practice.

One way to effectively train a neural network with multiple layers is by training one layer at a time. You can achieve this by training a special type of network known as an autoencoder for each desired hidden layer.

This example shows you how to train a neural network with two hidden layers to classify digits in images. First you train the hidden layers individually in an unsupervised fashion using autoencoders. Then you train a final soft max layer, and join the layers together to form a stacked network, which you train one final time in a supervised fashion.

Step 1 : Data Set

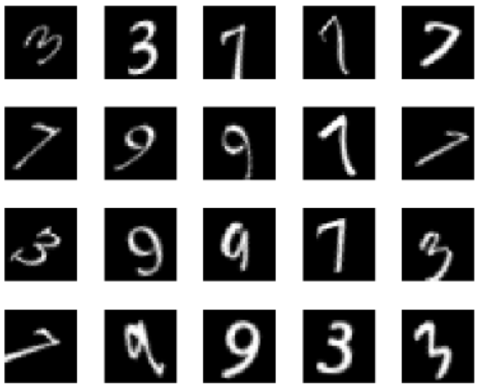
In the first step, you gather the data that you'll use to train your neural network. This data could come from various sources and may need preprocessing before being fed into the network. Data preprocessing may involve tasks such as normalization, standardization, handling missing values, and splitting the data into training and testing sets. The quality and quantity of your data significantly impact the performance and generalization ability of your neural network. Therefore, it's crucial to ensure that your dataset is representative of the problem you're trying to solve and is large enough to capture its complexity without overfitting.

This example uses synthetic data throughout, for training and testing. The synthetic images have been generated by applying random affine transformations to digit images created using different fonts. Each digit image is 28-by-28 pixels, and there are 5,000 training examples. You can load the training data, and view some of the images.

Code:

```
% Load the training data into memory

[xTrainImages,tTrain] = digitTrainCellArrayData;

% Display some of the training images

clf

for i = 1:20

   subplot(4,5,i);

   imshow(xTrainImages{i});

end
```

Output:



The labels for the images are arranged in a matrix of size 10-by-5000. In each column, only one element is set to 1, indicating the class to which the digit image belongs. All other elements in the column are set to 0. Notably, if the tenth element in a column is 1, it signifies that the digit image represents the number zero.

Step 2: Training the First Autoencoder

The second step involves training a neural network model known as an autoencoder. An autoencoder consists of an encoder and a decoder network. The encoder compresses the input data into a latent space representation, while the decoder reconstructs the original input from this representation. During training, the autoencoder learns to minimize the reconstruction error, effectively learning a compact representation of the input data. This step typically involves feeding the dataset into the autoencoder model iteratively, adjusting the model's parameters (weights and biases) using optimization techniques such as gradient descent, until the reconstruction error is minimized to an acceptable level Begin by training a sparse autoencoder on the training data without using the labels. An autoencoder is a neural network which attempts to replicate its input at its output. Thus, the size of its input will be the same as the size of its output. When the number of neurons in the hidden layer is less than the size of the input, the autoencoder learns a compressed representation of the input.

Neural networks have weights randomly initialized before training. Therefore the results from training are different each time. To avoid this behavior, explicitly

set the random number generator seed.

```
> rng('default')
```

Set the size of the hidden layer for the autoencoder. For the autoencoder that you are going to train, it is a good idea to make this smaller than the input size.

```
> hiddenSize1 = 100;
```

L2WeightRegularization should typically be quite small. SparsityRegularization controls the impact of a sparsity regularizer.

Note that this is different from applying a sparsity regularizer to the weights. Sparsity Proportion is a parameter of the sparsity regularizer.

It controls the sparsity of the output from the hidden layer. A low value for SparsityProportion usually leads to each neuron in the hidden layer "specializing" by only giving a high output for a small number of training examples.

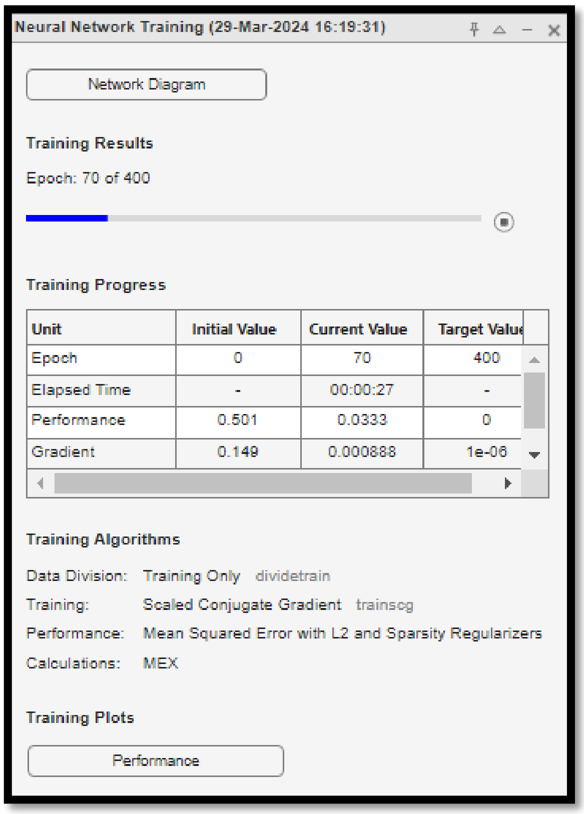
For example, if SparsityProportion is set to 0.1, this is equivalent to saying that each neuron in the hidden layer should have an average output of 0.1 over the training examples.

This value must be between 0 and 1. The ideal value varies depending on the nature of the problem.

Now train the autoencoder, specifying the values for the regularizers that are described above.

```
autoenc1 = trainAutoencoder(xTrainImages,hiddenSize1, ...
    'MaxEpochs',400, ...
    'L2WeightRegularization',0.004, ...
    'SparsityRegularization',4, ...
    'SparsityProportion',0.15, ...
    'ScaleData', false);
```

Training in Progress:



Neural Network Training (29-Mar-2024 16:19:31)

Network Diagram

Training Results

Epoch: 70 of 400

Training Progress

Unit	Initial Value	Current Value	Target Value
Epoch	0	70	400
Elapsed Time	-	00:00:27	-
Performance	0.501	0.0333	0
Gradient	0.149	0.000888	1e-06

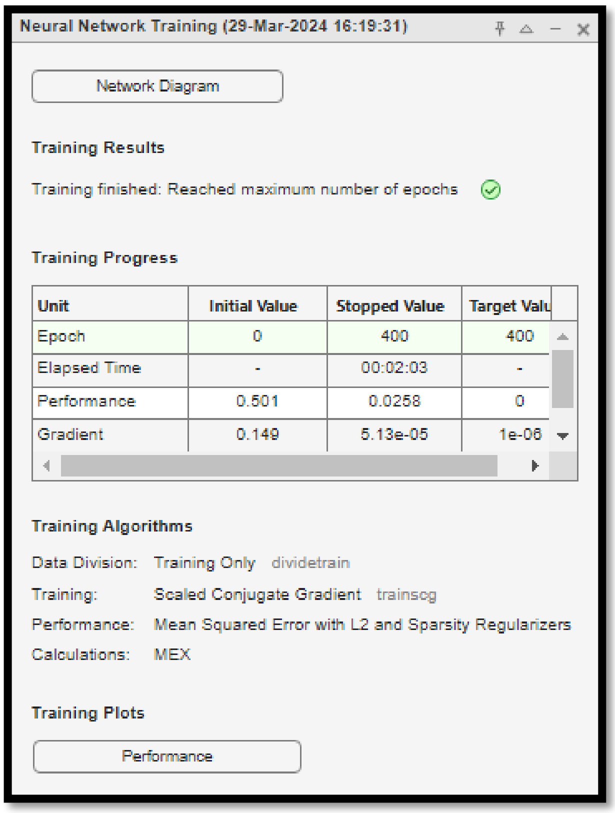
Training Algorithms

Data Division:	Training Only	dividetrain
Training:	Scaled Conjugate Gradient	trainscg
Performance:	Mean Squared Error with L2 and Sparsity Regularizers	
Calculations:	MEX	

Training Plots

Performance

Training Finished:



Neural Network Training (29-Mar-2024 16:19:31)

Network Diagram

Training Results

Training finished: Reached maximum number of epochs ✓

Training Progress

Unit	Initial Value	Stopped Value	Target Valu
Epoch	0	400	400
Elapsed Time	-	00:02:03	-
Performance	0.501	0.0258	0
Gradient	0.149	5.13e-05	1e-06

Training Algorithms

Data Division:	Training Only	dividetrain
Training:	Scaled Conjugate Gradient	trainscg
Performance:	Mean Squared Error with L2 and Sparsity Regularizers	
Calculations:	MEX	

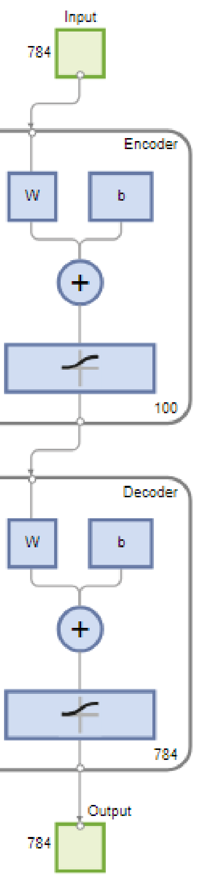
Training Plots

Performance

View Auto Encoder

To view a diagram of the autoencoder use this command:

> view(autoenc1)



<u>Step 3: Visualizing the weights of the first autoencoder</u>

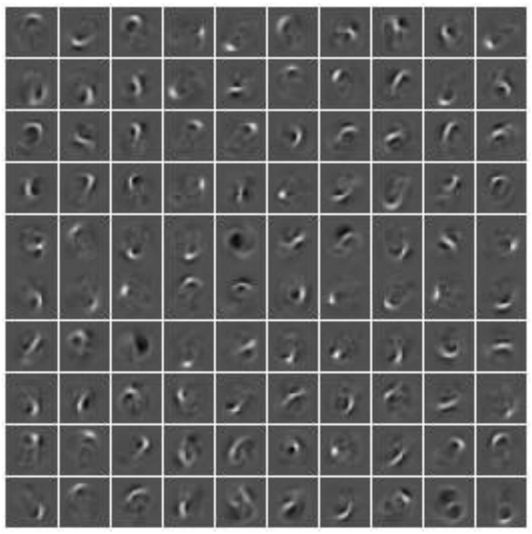
After training the first autoencoder, it's essential to understand what features or patterns the model has learned. One way to gain insights into the learned representations is by visualizing the weights of the auto encoder's layers. Visualizing weights can provide valuable information about the structure and complexity of the learned features. Techniques such as plotting histograms, heat maps, or feature maps can reveal which parts of the input space are most important for the model's reconstruction task. This step helps in debugging the model, identifying over fitting or under fitting issues, and gaining intuition about the learned representations, which can guide further model development and optimization.

The mapping learned by the encoder part of an autoencoder can be useful for extracting features from data.

Each neuron in the encoder has a vector of weights associated with it which will be tuned to respond to a particular visual feature.

You can view a representation of these features.

```
> figure()
>plotWeights(autoenc1);
```



The features learned by the autoencoder capture patterns like curls and strokes found in the digit images. The output from the hidden layer, which is 100-dimensional, gives a condensed version of the input data. This compressed representation summarizes how the autoencoder responds to the visual features it has learned. To train the next autoencoder, we'll use these extracted feature vectors from the training data. Initially, we'll employ the encoder part of the trained autoencoder to generate these features.

```
> feat1 = encode(autoenc1,xTrainImages);
```

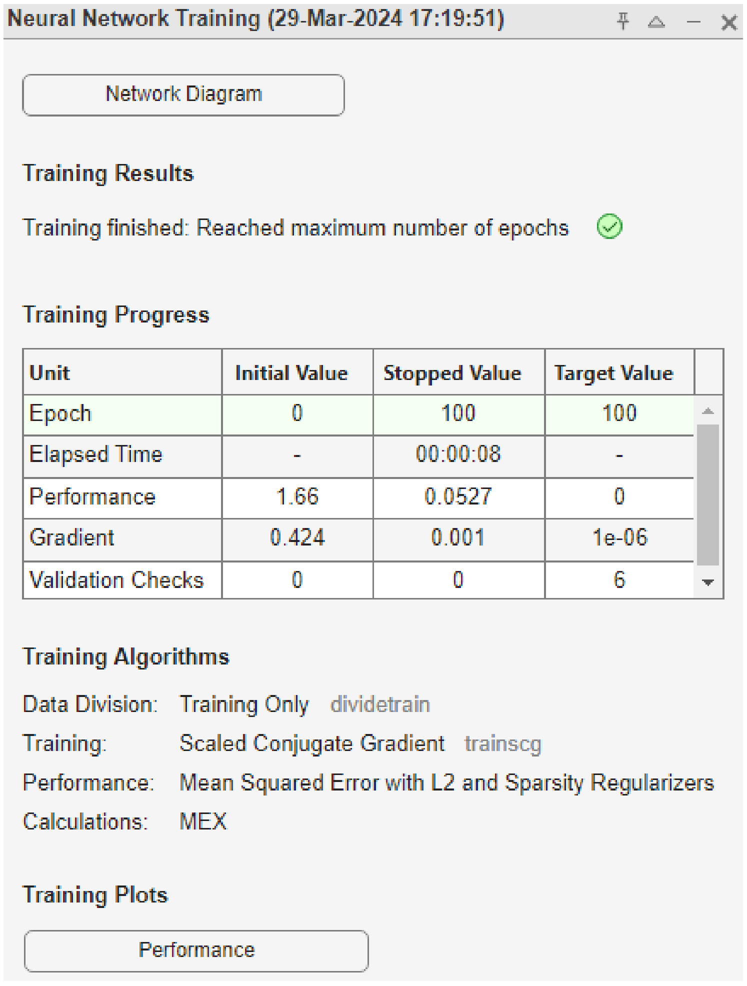
Step 4: Visualizing Training the second autoencoder

After obtaining the compressed feature representations from the first autoencoder, the next step is to train a second autoencoder. This autoencoder follows a similar architecture as the first one, comprising an encoder and a decoder. However, this time, the input to the second autoencoder is the compressed feature vectors extracted from the training data using the encoder of the first autoencoder. By training the second autoencoder, the model learns to further compress and reconstruct these feature representations. During training, the second autoencoder adjusts its parameters to minimize the reconstruction error, aiming to reconstruct the original feature vectors as accurately as possible. This step is crucial for refining the learned representations and capturing more abstract and higher-level features present in the data.

After training the first autoencoder, you train the second autoencoder in a similar way. The main difference is that you use the features that were generated from the first autoencoder as the training data in the second autoencoder.

Also, you decrease the size of the hidden representation to 50, so that the encoder in the second autoencoder learns an even smaller representation of the input data.

```
hiddenSize2 = 50;

autoenc2 = trainAutoencoder(feat1,hiddenSize2, ...

  'MaxEpochs',100, ...

  'L2WeightRegularization',0.002, ...

  'SparsityRegularization',4, ...

  'SparsityProportion',0.1, ...

  'ScaleData', false);
```



Neural Network Training (29-Mar-2024 17:19:51)

Training Results

Training finished: Reached maximum number of epochs ✓

Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	100	100
Elapsed Time	-	00:00:08	-
Performance	1.66	0.0527	0
Gradient	0.424	0.001	1e-06
Validation Checks	0	0	6

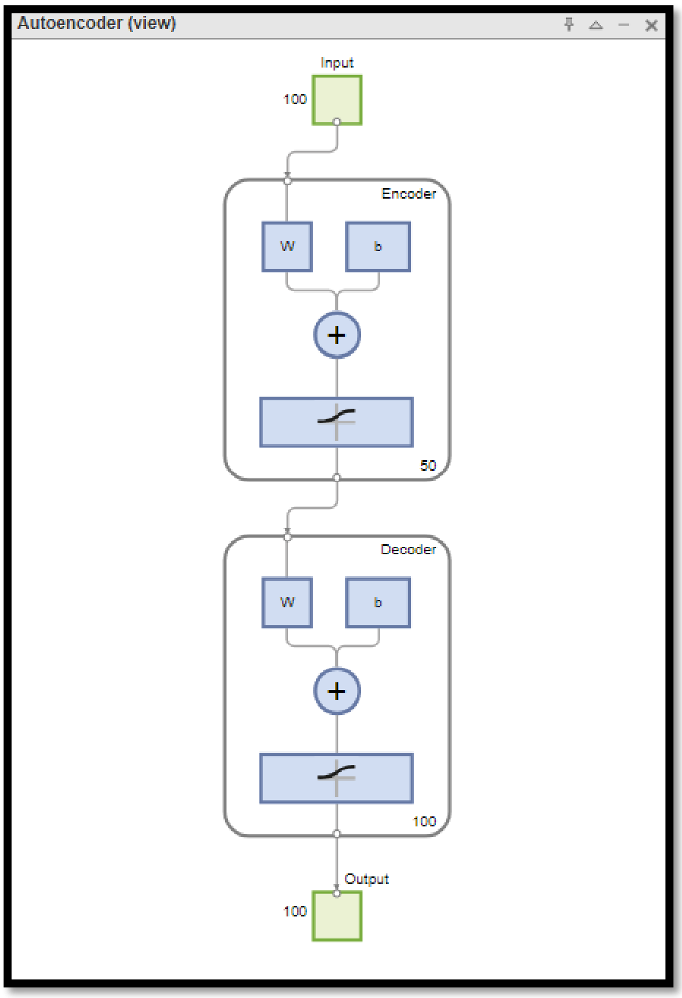
Training Algorithms

Data Division:	Training Only	dividetrain
Training:	Scaled Conjugate Gradient	trainscg
Performance:	Mean Squared Error with L2 and Sparsity Regularizers	
Calculations:	MEX	

Training Plots

For the Re Verification Once again, you can view a diagram of the autoencoder with the view function.

> view(autoenc2)



Another set of features can be derived by running the previous set through the encoder of the second autoencoder.

> feat2 = encode(autoenc2,feat1);

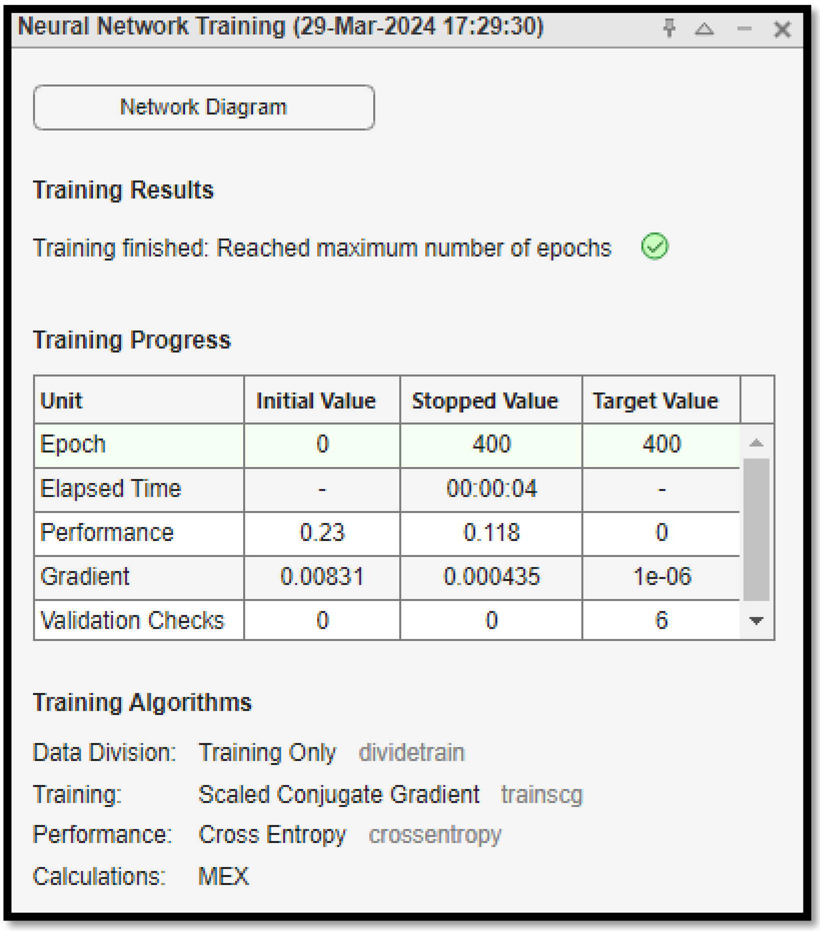
The initial vectors in the training dataset had 784 dimensions. Following their transformation through the first encoder, this dimensionality was reduced to 100. Subsequently, utilizing the second encoder further reduced this dimensionality to 50. Now, the opportunity arises to train a concluding layer aimed at classifying these 50-dimensional vectors into distinct digit classes.

Step 5: Training the Final Soft Max Layer

Once the second autoencoder is trained, the next step is to add a final softmax layer on top of the stacked autoencoder architecture. The softmax layer is a common choice for multiclass classification tasks. It takes the output of the stacked autoencoder, which represents the compressed and refined features, and applies the softmax function to compute the probabilities of the input belonging to each class. During training, the parameters of the softmax layer are optimized using techniques like gradient descent to minimize the categorical cross-entropy loss between the predicted probabilities and the true labels. This step essentially transforms the learned features into class probabilities, enabling the model to make predictions on new unseen data.

Train a softmax layer to categorize the 50-dimensional feature vectors. In contrast to the autoencoders, this training process for the softmax layer is supervised, utilizing labeled data from the training dataset.

```
> softnet = trainSoftmaxLayer(feat2,tTrain,'MaxEpochs',400);
```



Neural Network Training (29-Mar-2024 17:29:30)

Training Results

Training finished: Reached maximum number of epochs ✓

Training Progress

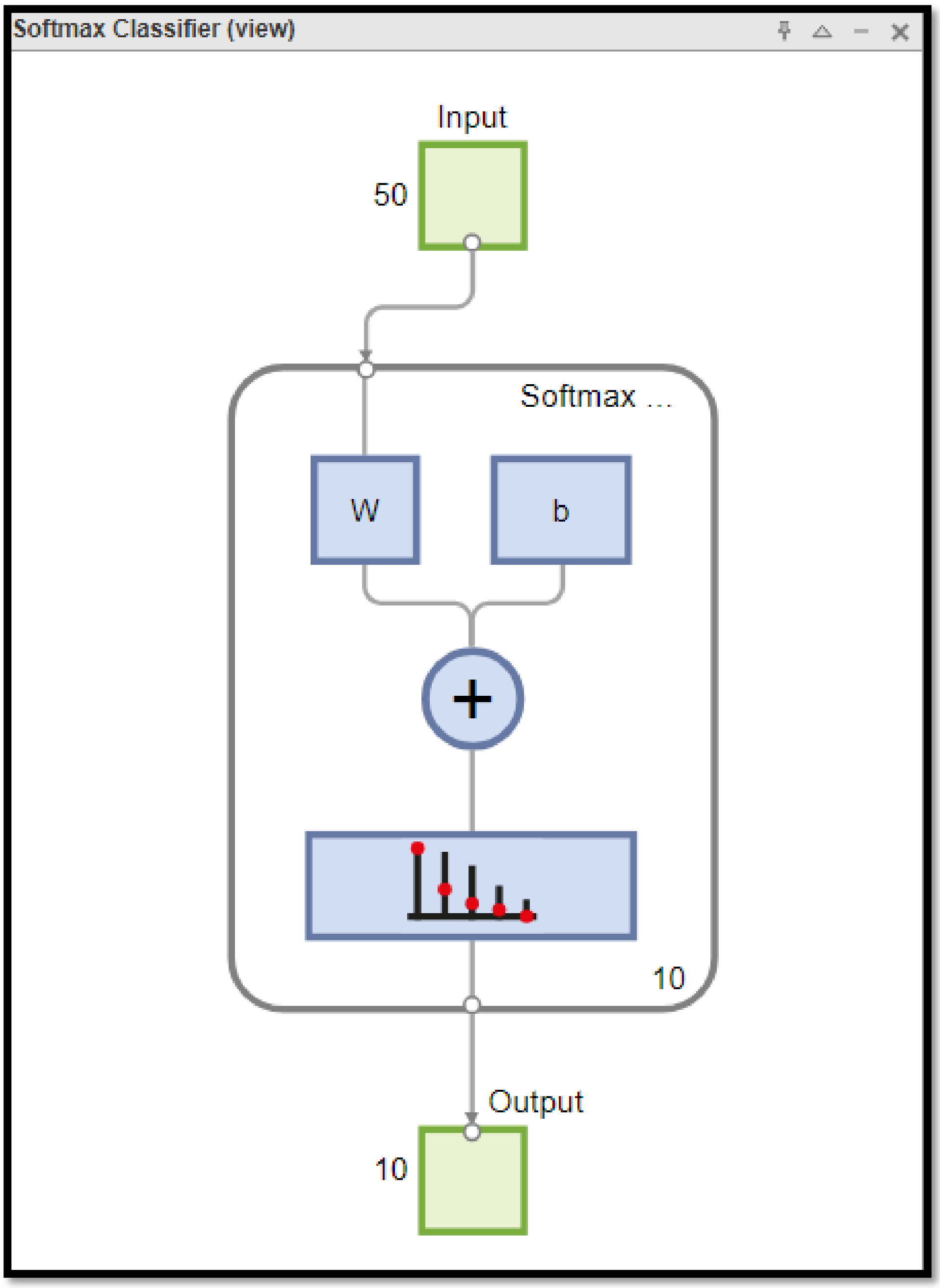
Unit	Initial Value	Stopped Value	Target Value
Epoch	0	400	400
Elapsed Time	-	00:00:04	-
Performance	0.23	0.118	0
Gradient	0.00831	0.000435	1e-06
Validation Checks	0	0	6

Training Algorithms

Data Division:	Training Only	dividetrain
Training:	Scaled Conjugate Gradient	trainscg
Performance:	Cross Entropy	crossentropy
Calculations:	MEX	

You can view a diagram of the softmax layer with the view function.

```
> view(softnet)
```



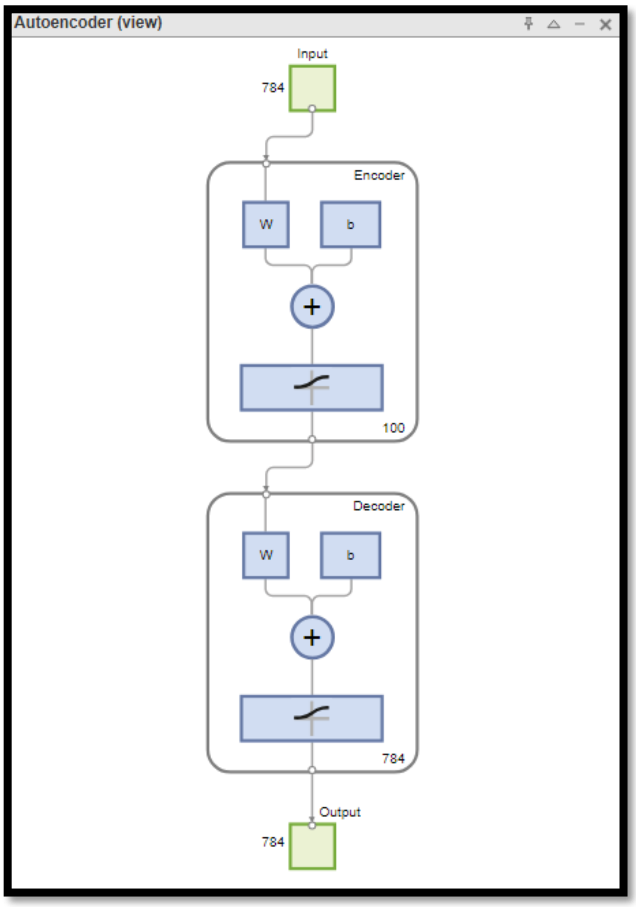
Step 6: Forming a stacked neural network

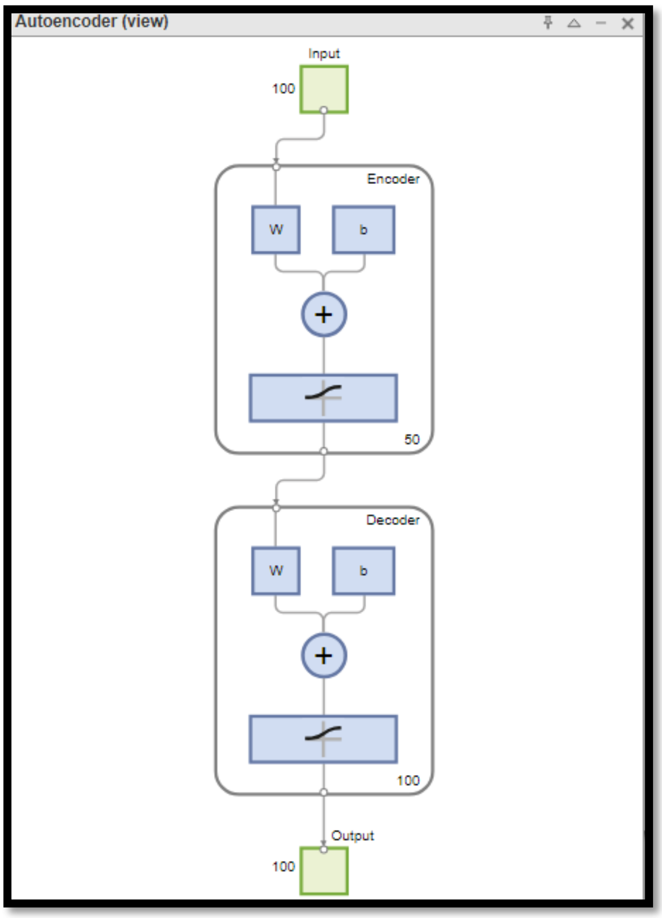
With the second autoencoder trained and the final softmax layer added, the next step is to form a stacked neural network by combining these components. The stacked neural network consists of the encoder part of the first autoencoder, followed by the encoder of the second autoencoder, and finally, the softmax layer. This architecture forms a hierarchical feature extractor, where the first autoencoder learns basic features, the second autoencoder learns more abstract features based on the compressed representations from the first autoencoder, and the softmax layer performs classification based on these learned features. Training the stacked neural network involves fine-tuning the entire model using back propagation with techniques such as stochastic gradient descent (SGD) or Adam optimization. Fine-tuning ensures that all components of the stacked neural network work together effectively to achieve the desired classification performance.

You have trained three separate components of a stacked neural network in isolation. At this point, it might be useful to view the three neural networks that you have trained. They are autoenc1, autoenc2, and softnet

view(autoenc1)

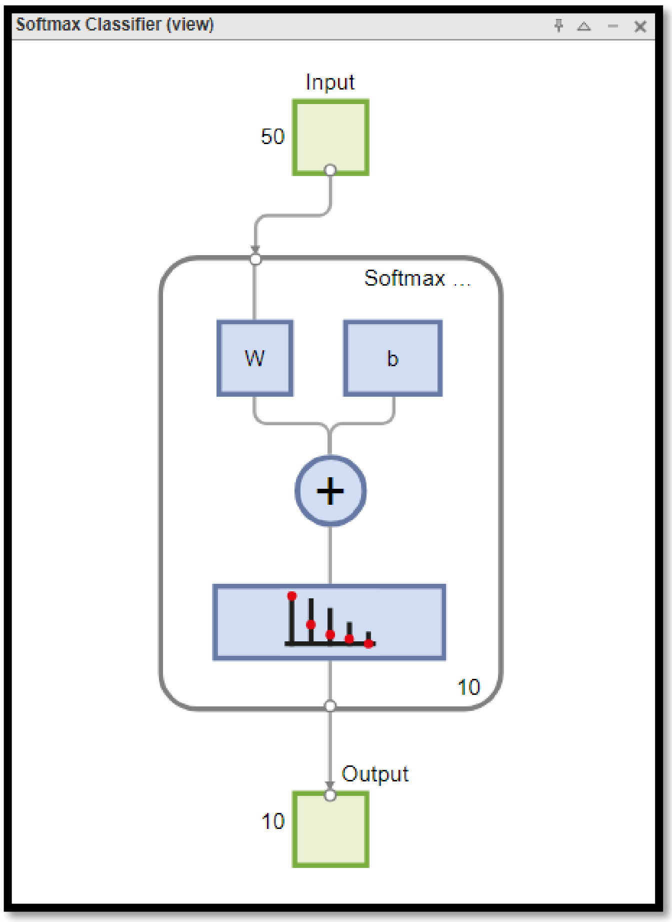
view(autoenc2)

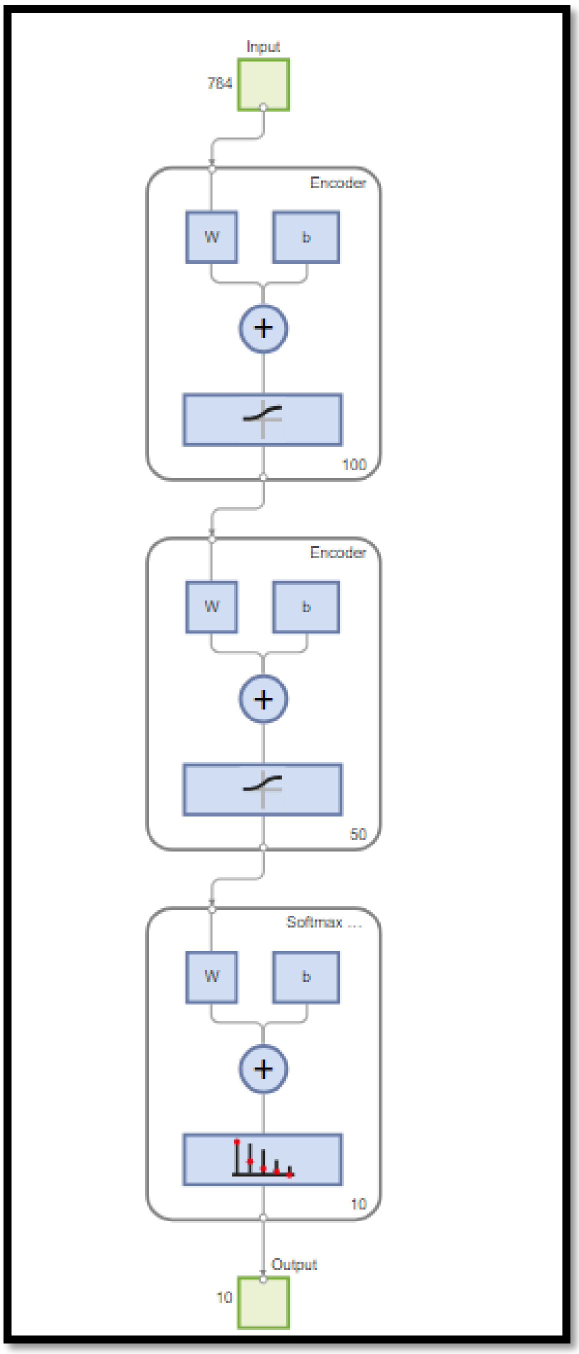




view(softnet)

view(stackednet)





As previously described, the encoders from the autoencoders have been employed to extract features. Combining the encoders from the autoencoders with the softmax layer forms a stacked network for classification. This amalgamation is achieved by stacking the encoders from the autoencoders along with the softmax layer. To visualize the architecture of the stacked network, you can utilize the view function. The resultant network comprises the encoders from the autoencoders and the softmax layer.

Commands to be used:

```
> stackednet = stack(autoenc1,autoenc2,softnet);
> view(stackednet)
```

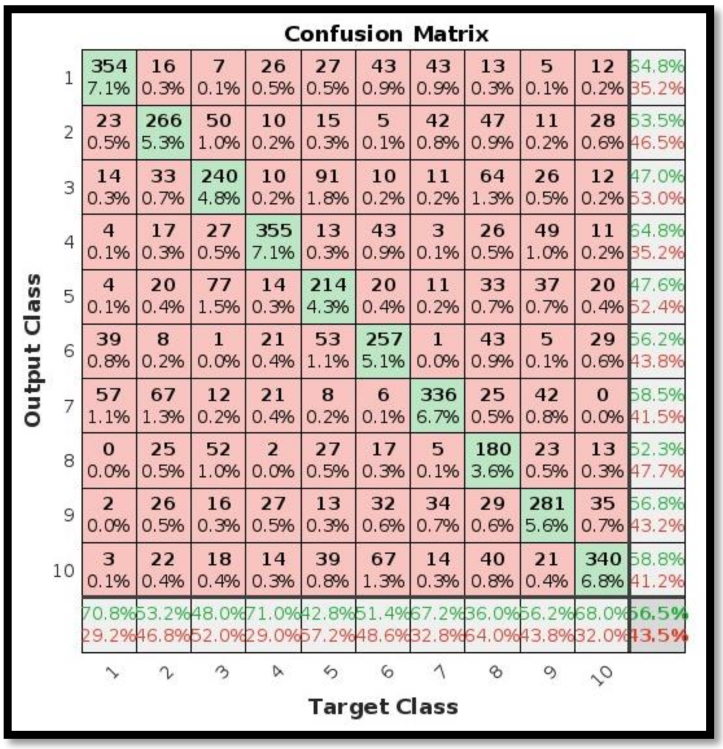
Now that the entire network is assembled, you can evaluate its performance on the test set. However, before utilizing the images with the stacked network, it's necessary to reshape the test images into a matrix format. This can be accomplished by arranging the columns of an image into a vector, and then constructing a matrix from these vectors.

Code

```
% Get the number of pixels in each image

imageWidth = 28;

imageHeight = 28;

inputSize = imageWidth*imageHeight;

% Load the test images

[xTestImages,tTest] = digitTestCellArrayData;

% Turn the test images into vectors and put them in a matrix

xTest = zeros(inputSize,numel(xTestImages));

for i = 1:numel(xTestImages)

   xTest(:,i) = xTestImages{i}(:);

end
```

You can visualize the results with a confusion matrix. The numbers in the bottom right-hand square of the matrix give the overall accuracy.

```
> y = stackednet(xTest);

>plotconfusion(tTest,y);
```



Confusion Matrix

	1	2	3	4	5	6	7	8	9	10	
1	354 7.1%	16 0.3%	7 0.1%	26 0.5%	27 0.5%	43 0.9%	43 0.9%	13 0.3%	5 0.1%	12 0.2%	64.8% 35.2%
2	23 0.5%	266 5.3%	50 1.0%	10 0.2%	15 0.3%	5 0.1%	42 0.8%	47 0.9%	11 0.2%	28 0.6%	53.5% 46.5%
3	14 0.3%	33 0.7%	240 4.8%	10 0.2%	91 1.8%	10 0.2%	11 0.2%	64 1.3%	26 0.5%	12 0.2%	47.0% 53.0%
4	4 0.1%	17 0.3%	27 0.5%	355 7.1%	13 0.3%	43 0.9%	3 0.1%	26 0.5%	49 1.0%	11 0.2%	64.8% 35.2%
5	4 0.1%	20 0.4%	77 1.5%	14 0.3%	214 4.3%	20 0.4%	11 0.2%	33 0.7%	37 0.7%	20 0.4%	47.6% 52.4%
6	39 0.8%	8 0.2%	1 0.0%	21 0.4%	53 1.1%	257 5.1%	1 0.0%	43 0.9%	5 0.1%	29 0.6%	56.2% 43.8%
7	57 1.1%	67 1.3%	12 0.2%	21 0.4%	8 0.2%	6 0.1%	336 6.7%	25 0.5%	42 0.8%	0 0.0%	58.5% 41.5%
8	0 0.0%	25 0.5%	52 1.0%	2 0.0%	27 0.5%	17 0.3%	5 0.1%	180 3.6%	23 0.5%	13 0.3%	52.3% 47.7%
9	2 0.0%	26 0.5%	16 0.3%	27 0.5%	13 0.3%	32 0.6%	34 0.7%	29 0.6%	281 5.6%	35 0.7%	56.8% 43.2%
10	3 0.1%	22 0.4%	18 0.4%	14 0.3%	39 0.8%	67 1.3%	14 0.3%	40 0.8%	21 0.4%	340 6.8%	58.8% 41.2%
	70.8% 29.2%	53.2% 46.8%	48.0% 52.0%	71.0% 29.0%	42.8% 57.2%	51.4% 48.6%	67.2% 32.8%	36.0% 64.0%	56.2% 43.8%	68.0% 32.0%	56.5% 43.5%

Step 7: Fine-Tuning the Stacked Neural Network

Fine-tuning the stacked neural network involves adjusting the parameters of the entire model, including the encoders from the autoencoders and the softmax layer, to optimize its performance on the task at hand, which in this case is digit classification. This step is crucial for refining the learned representations and improving the overall accuracy of the model. Fine-tuning typically involves using techniques like back propagation with adaptive optimization algorithms such as stochastic gradient descent (SGD) or Adam to update the weights and biases of the network. During fine-tuning, the model learns to better discriminate between different digit classes by iteratively adjusting its parameters based on the gradients of the loss function with respect to these parameters.

The results for the stacked neural network can be improved by performing backpropagation on the whole multilayer network. This process is often referred to as fine tuning.

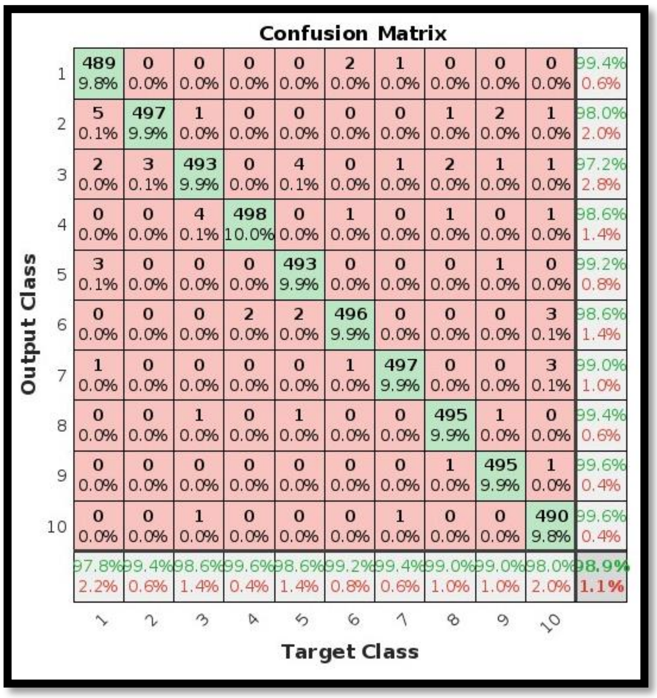
You fine tune the network by retraining it on the training data in a supervised fashion. Before you can do this, you have to reshape the training images into a matrix, as was done for the test images.

Code

```
% Turn the training images into vectors and put them in a matrix

xTrain = zeros(inputSize,numel(xTrainImages));

for i = 1:numel(xTrainImages)

   xTrain(:,i) = xTrainImages{i}(:);

end

% Perform fine tuning

stackednet = train(stackednet,xTrain,tTrain);
```

You then view the results again using a confusion matrix.

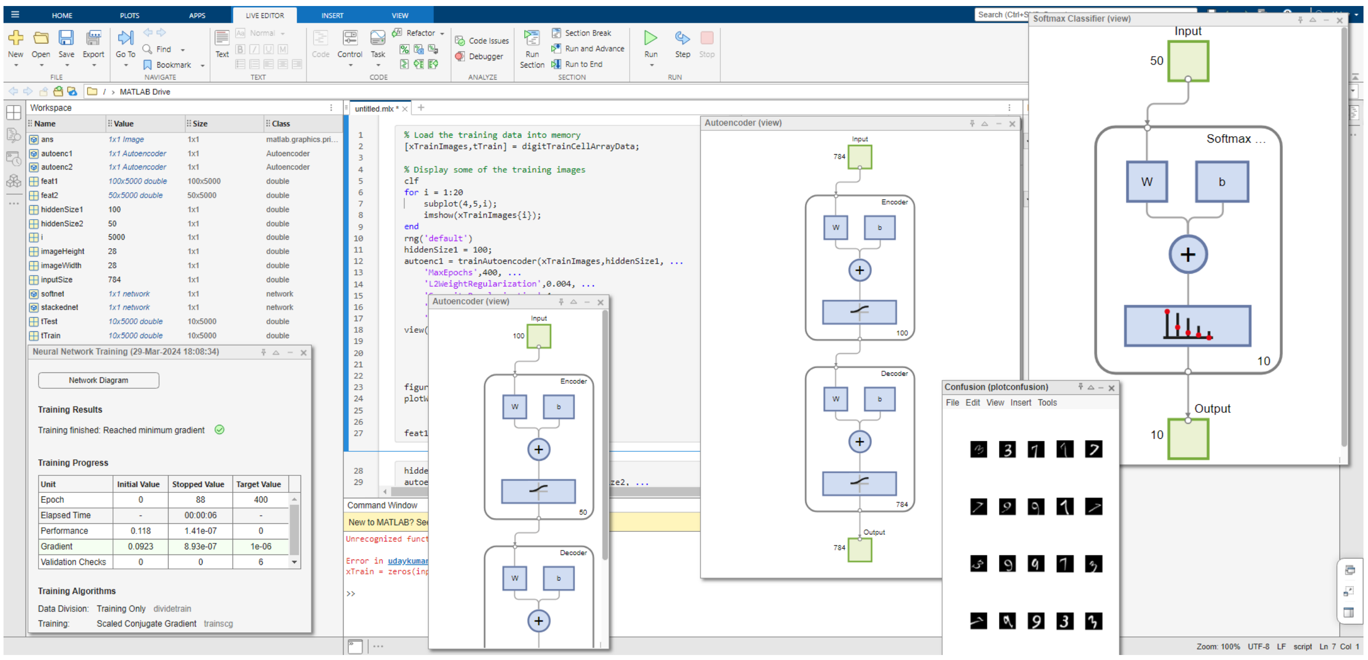
```
> y = stackednet(xTest);

>plotconfusion(tTest,y);
```



Confusion Matrix

Output Class \ Target Class	1	2	3	4	5	6	7	8	9	10	
1	489 9.8%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	2 0.0%	1 0.0%	0 0.0%	0 0.0%	0 0.0%	99.4% 0.6%
2	5 0.1%	497 9.9%	1 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	2 0.0%	1 0.0%	98.0% 2.0%
3	2 0.0%	3 0.1%	493 9.9%	0 0.0%	4 0.1%	0 0.0%	1 0.0%	2 0.0%	1 0.0%	1 0.0%	97.2% 2.8%
4	0 0.0%	0 0.0%	4 0.1%	498 10.0%	0 0.0%	1 0.0%	0 0.0%	1 0.0%	0 0.0%	1 0.0%	98.6% 1.4%
5	3 0.1%	0 0.0%	0 0.0%	0 0.0%	493 9.9%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	0 0.0%	99.2% 0.8%
6	0 0.0%	0 0.0%	0 0.0%	2 0.0%	2 0.0%	496 9.9%	0 0.0%	0 0.0%	0 0.0%	3 0.1%	98.6% 1.4%
7	1 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	497 9.9%	0 0.0%	0 0.0%	3 0.1%	99.0% 1.0%
8	0 0.0%	0 0.0%	1 0.0%	0 0.0%	1 0.0%	0 0.0%	0 0.0%	495 9.9%	1 0.0%	0 0.0%	99.4% 0.6%
9	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	495 9.9%	1 0.0%	99.6% 0.4%
10	0 0.0%	0 0.0%	1 0.0%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	0 0.0%	0 0.0%	490 9.8%	99.6% 0.4%
	97.8% 2.2%	99.4% 0.6%	98.6% 1.4%	99.6% 0.4%	98.6% 1.4%	99.2% 0.8%	99.4% 0.6%	99.0% 1.0%	99.0% 1.0%	98.0% 2.0%	98.9% 1.1%

Complete Work Interface:



Workspace

Name	Value	Size	Class
ans	*1x1 Image*	1x1	matlab.graphics.pri...
autoenc1	*1x1 Autoencoder*	1x1	Autoencoder
autoenc2	*1x1 Autoencoder*	1x1	Autoencoder
feat1	*100x5000 double*	100x5000	double
feat2	*50x5000 double*	50x5000	double
hiddenSize1	100	1x1	double
hiddenSize2	50	1x1	double
i	5000	1x1	double
imageHeight	28	1x1	double
imageWidth	28	1x1	double
inputSize	784	1x1	double
softnet	*1x1 network*	1x1	network
stackednet	*1x1 network*	1x1	network
tTest	*10x5000 double*	10x5000	double
tTrain	*10x5000 double*	10x5000	double

untitled.mlx *

```
% Load the training data into memory
[xTrainImages,tTrain] = digitTrainCellArrayData;

% Display some of the training images
clf
for i = 1:20
    subplot(4,5,i);
    imshow(xTrainImages{i});
end
rng('default')
hiddenSize1 = 100;
autoenc1 = trainAutoencoder(xTrainImages,hiddenSize1, ...
    'MaxEpochs',400, ...
    'L2WeightRegularization',0.004, ...
```

Neural Network Training (29-Mar-2024 18:08:34)

Training Results

Training finished: Reached minimum gradient ✓

Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	88	400
Elapsed Time	-	00:00:06	-
Performance	0.118	1.41e-07	0
Gradient	0.0923	8.93e-07	1e-06
Validation Checks	0	0	6

Training Algorithms

Data Division: Training Only dividetrain

Training: Scaled Conjugate Gradient trainscg

Command Window

```
Unrecognized funct

Error in udaykumar
xTrain = zeros(inp

>>
```

Summary

In summary, fine-tuning the stacked neural network encompasses optimizing the entire model by adjusting its parameters to enhance its ability to classify digit images accurately.

This process leverages the learned representations from the autoencoders and the classification capabilities of the softmax layer to refine the model's performance.

By iteratively updating the weights and biases of the network using backpropagation and optimization algorithms, the model learns to better capture the underlying patterns and features in the data, ultimately leading to improved classification accuracy on both training and test datasets.

MATLAB Link :

https://drive.matlab.com/sharing/6741adb0-1b25-4207-8169-cb08354ffeeb

Case Study 2 : Train the stacked auto-encoders for the classification of images.

Name : B Uday Kumar

Pin : HU21CSEN0100964

Neural networks with multiple hidden layers can be useful for solving classification problems with complex data, such as images. Each layer can learn features at a different level of abstraction. However, training neural networks with multiple hidden layers can be difficult in practice.

One way to effectively train a neural network with multiple layers is by training one layer at a time. You can achieve this by training a special type of network known as an autoencoder for each desired hidden layer.

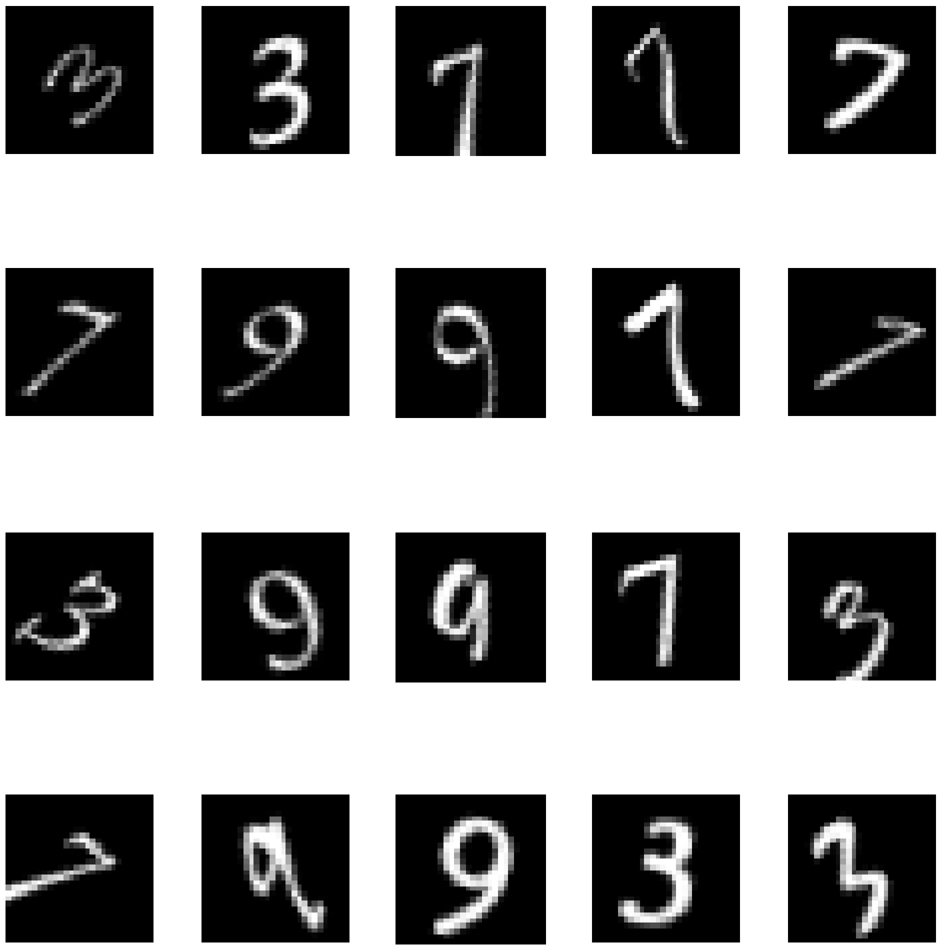
This example shows you how to train a neural network with two hidden layers to classify digits in images. First you train the hidden layers individually in an unsupervised fashion using autoencoders. Then you train a final soft max layer, and join the layers together to form a stacked network, which you train one final time in a supervised fashion.

Step 1 : Data Set

In the first step, you gather the data that you'll use to train your neural network. This data could come from various sources and may need preprocessing before being fed into the network. Data preprocessing may involve tasks such as normalization, standardization, handling missing values, and splitting the data into training and testing sets. The quality and quantity of your data significantly impact the performance and generalization ability of your neural network. Therefore, it's crucial to ensure that your dataset is representative of the problem you're trying to solve and is large enough to capture its complexity without overfitting.

This example uses synthetic data throughout, for training and testing. The synthetic images have been generated by applying random affine transformations to digit images created using different fonts. Each digit image is 28-by-28 pixels, and there are 5,000 training examples. You can load the training data, and view some of the images.

```
% Load the training data into memory
[xTrainImages,tTrain] = digitTrainCellArrayData;

% Display some of the training images
clf
for i = 1:20
    subplot(4,5,i);
    imshow(xTrainImages{i});
end
```



Neural networks have weights randomly initialized before training. Therefore the results from training are different each time. To avoid this behavior, explicitly

set the random number generator seed.

```
rng('default')
```

Set the size of the hidden layer for the autoencoder. For the autoencoder that you are going to train, it is a good idea to make this smaller than the input size.

```
hiddenSize1 = 100;
```

L2WeightRegularization should typically be quite small. SparsityRegularization controls the impact of a sparsity regularizer.

Note that this is different from applying a sparsity regularizer to the weights. Sparsity Proportion is a parameter of the sparsity regularizer.

It controls the sparsity of the output from the hidden layer. A low value for SparsityProportion usually leads to each neuron in the hidden layer "specializing" by only giving a high output for a small number of training examples.

For example, if SparsityProportion is set to 0.1, this is equivalent to saying that each neuron in the hidden layer should have an average output of 0.1 over the training examples.

This value must be between 0 and 1. The ideal value varies depending on the nature of the problem.

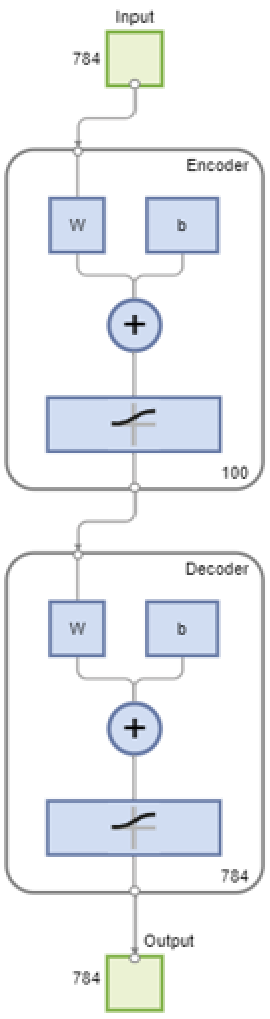
Now train the autoencoder, specifying the values for the regularizers that are described above.

```
autoenc1 = trainAutoencoder(xTrainImages,hiddenSize1, ...
    'MaxEpochs',400, ...
    'L2WeightRegularization',0.004, ...
    'SparsityRegularization',4, ...
    'SparsityProportion',0.15, ...
    'ScaleData', false);
```

View Auto Encoder

To view a diagram of the autoencoder use this command:

```
view(autoenc1)
```



Step 3: Visualizing the weights of the first autoencoder

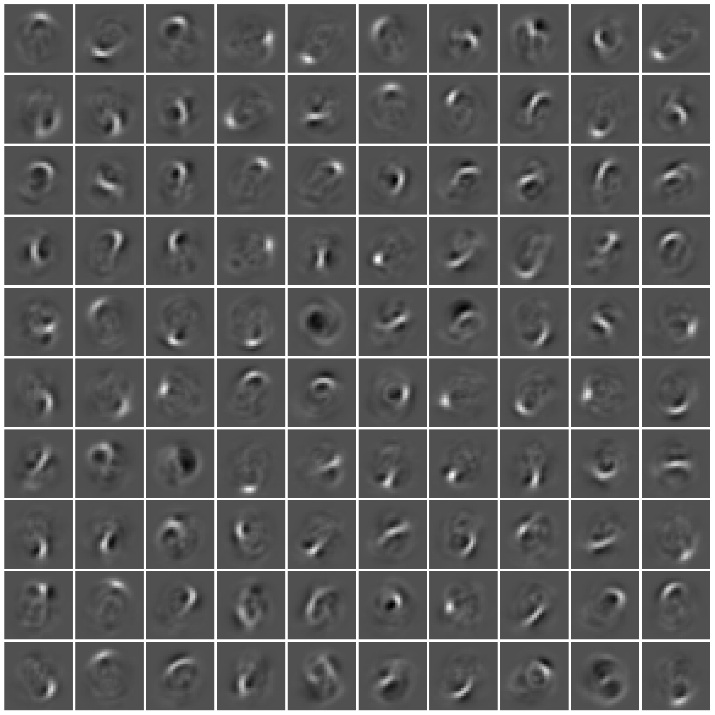
After training the first autoencoder, it's essential to understand what features or patterns the model has learned. One way to gain insights into the learned representations is by visualizing the weights of the auto encoder's layers. Visualizing weights can provide valuable information about the structure and complexity of the learned features. Techniques such as plotting histograms, heat maps, or feature maps can reveal which parts of the input space are most important for the model's reconstruction task. This step helps in debugging the model, identifying over fitting or under fitting issues, and gaining intuition about the learned representations, which can guide further model development and optimization.

The mapping learned by the encoder part of an autoencoder can be useful for extracting features from data.

Each neuron in the encoder has a vector of weights associated with it which will be tuned to respond to a particular visual feature.

You can view a representation of these features.

```
figure()
plotWeights(autoenc1);
```



The features learned by the autoencoder capture patterns like curls and strokes found in the digit images. The output from the hidden layer, which is 100-dimensional, gives a condensed version of the input data. This compressed representation summarizes how the autoencoder responds to the visual features it has learned. To train the next autoencoder, we'll use these extracted feature vectors from the training data. Initially, we'll employ the encoder part of the trained autoencoder to generate these features.

```
feat1 = encode(autoenc1,xTrainImages);
```

Step 4: Visualizing Training the second autoencoder

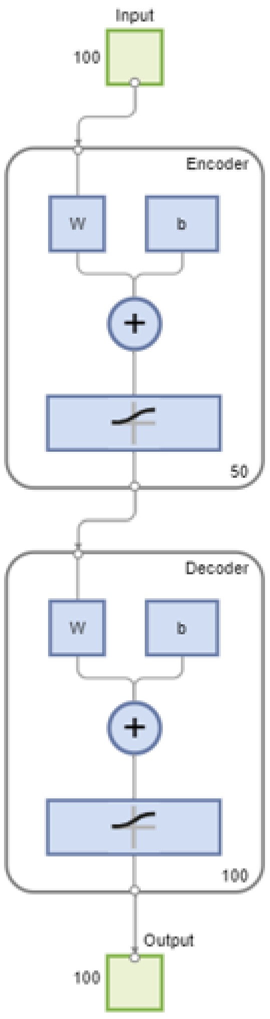
After obtaining the compressed feature representations from the first autoencoder, the next step is to train a second autoencoder. This autoencoder follows a similar architecture as the first one, comprising an encoder and a decoder. However, this time, the input to the second autoencoder is the compressed feature vectors extracted from the training data using the encoder of the first autoencoder. By training the second autoencoder, the model learns to further compress and reconstruct these feature representations. During training, the second autoencoder adjusts its parameters to minimize the reconstruction error, aiming to reconstruct the original feature vectors as accurately as possible. This step is crucial for refining the learned representations and capturing more abstract and higher-level features present in the data.

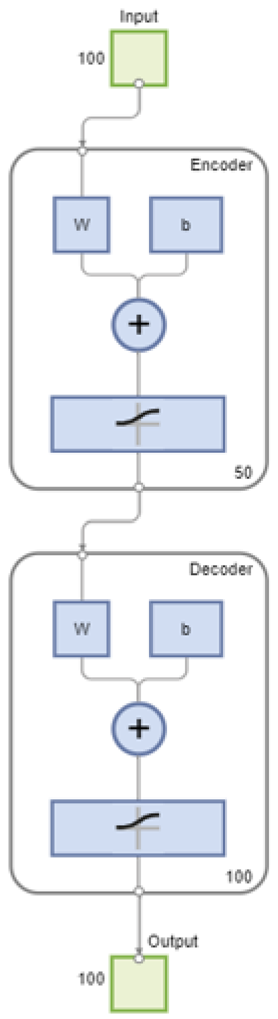
After training the first autoencoder, you train the second autoencoder in a similar way. The main difference is that you use the features that were generated from the first autoencoder as the training data in the second autoencoder. Also, you decrease the size of the hidden representation to 50, so that the encoder in the second autoencoder learns an even smaller representation of the input data.

```
hiddenSize2 = 50;
autoenc2 = trainAutoencoder(feat1,hiddenSize2, ...
    'MaxEpochs',100, ...
    'L2WeightRegularization',0.002, ...
    'SparsityRegularization',4, ...
    'SparsityProportion',0.1, ...
    'ScaleData', false);
```

For the Re Verification Once again, you can view a diagram of the autoencoder with the view function

```
view(autoenc2)
```





Another set of features can be derived by running the previous set through the encoder of the second autoencoder.

```
feat2 = encode(autoenc2,feat1);
```

The initial vectors in the training dataset had 784 dimensions. Following their transformation through the first encoder, this dimensionality was reduced to 100. Subsequently, utilizing the second encoder further reduced this dimensionality to 50. Now, the opportunity arises to train a concluding layer aimed at classifying these 50-dimensional vectors into distinct digit classes.

Step 5: Training the Final Soft Max Layer

Once the second autoencoder is trained, the next step is to add a final softmax layer on top of the stacked autoencoder architecture. The softmax layer is a common choice for multiclass classification tasks. It takes the output of the stacked autoencoder, which represents the compressed and refined features, and applies the softmax function to compute the probabilities of the input belonging to each class. During training, the parameters of the softmax layer are optimized using techniques like gradient descent to minimize the categorical cross-entropy loss between the predicted probabilities and the true labels. This step essentially transforms the learned features into class probabilities, enabling the model to make predictions on new unseen data.

Train a softmax layer to categorize the 50-dimensional feature vectors. In contrast to the autoencoders, this training process for the softmax layer is supervised, utilizing labeled data from the training dataset.

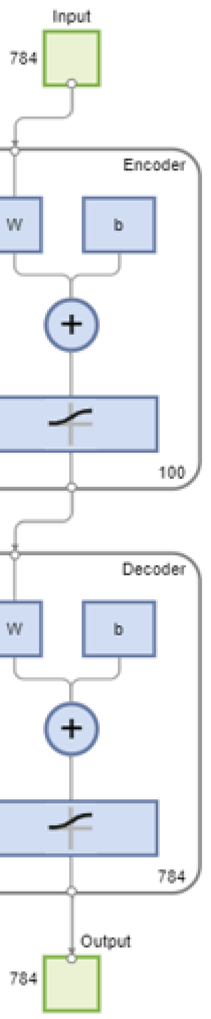
```
softnet = trainSoftmaxLayer(feat2,tTrain,'MaxEpochs',400);
```

Step 6: Forming a stacked neural network

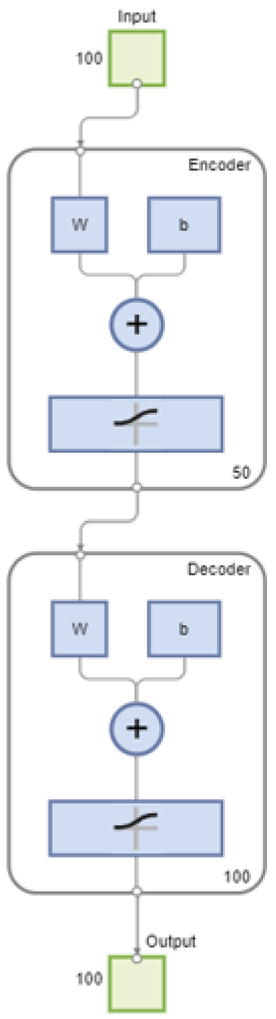
With the second autoencoder trained and the final softmax layer added, the next step is to form a stacked neural network by combining these components. The stacked neural network consists of the encoder part of the first autoencoder, followed by the encoder of the second autoencoder, and finally, the softmax layer. This architecture forms a hierarchical feature extractor, where the first autoencoder learns basic features, the second autoencoder learns more abstract features based on the compressed representations from the first autoencoder, and the softmax layer performs classification based on these learned features. Training the stacked neural network involves fine-tuning the entire model using back propagation with techniques such as stochastic gradient descent (SGD) or Adam optimization. Fine-tuning ensures that all components of the stacked neural network work together effectively to achieve the desired classification performance.

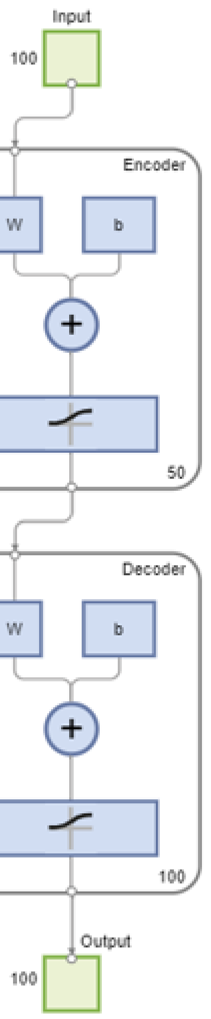
You have trained three separate components of a stacked neural network in isolation. At this point, it might be useful to view the three neural networks that you have trained. They are autoenc1, autoenc2, and softnet

```
view(autoenc1)
```

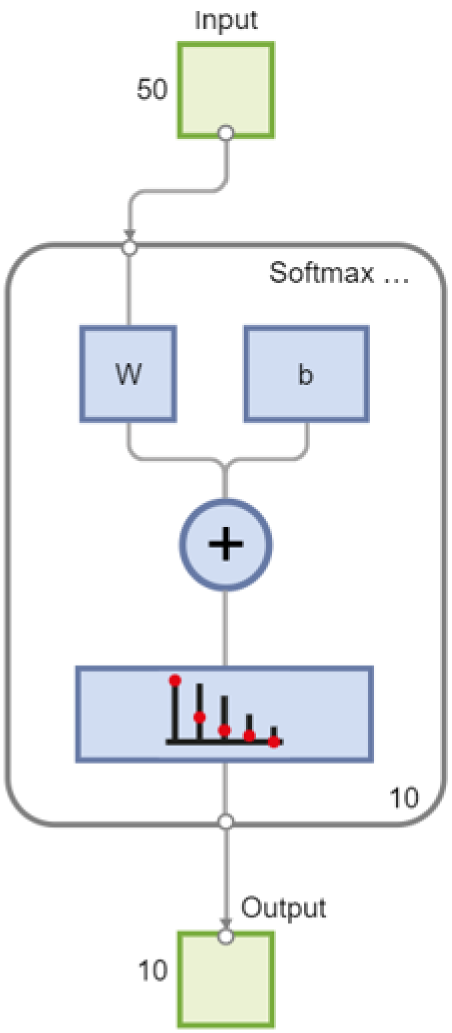


```
view(autoenc2)
```


Input
100
Encoder
W
b
50
Decoder
W
b
100
Output
100



```
view(softnet)
```

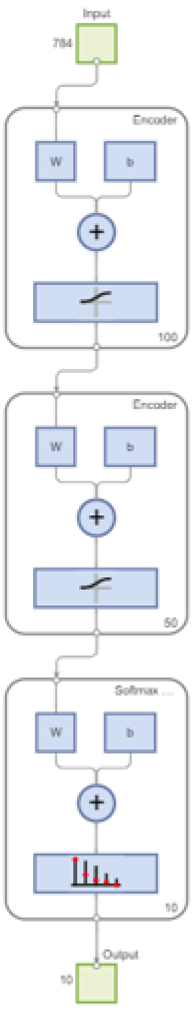


As previously described, the encoders from the autoencoders have been employed to extract features. Combining the encoders from the autoencoders with the softmax layer forms a stacked network for classification. This amalgamation is achieved by stacking the encoders from the autoencoders along with the softmax layer. To visualize the architecture of the stacked network, you can utilize the view function. The resultant network comprises the encoders from the autoencoders and the softmax layer.

```
stackednet = stack(autoenc1,autoenc2,softnet);
```

You can view a diagram of the stacked network with the `view` function. The network is formed by the encoders from the autoencoders and the softmax layer.

```
view(stackednet)
```



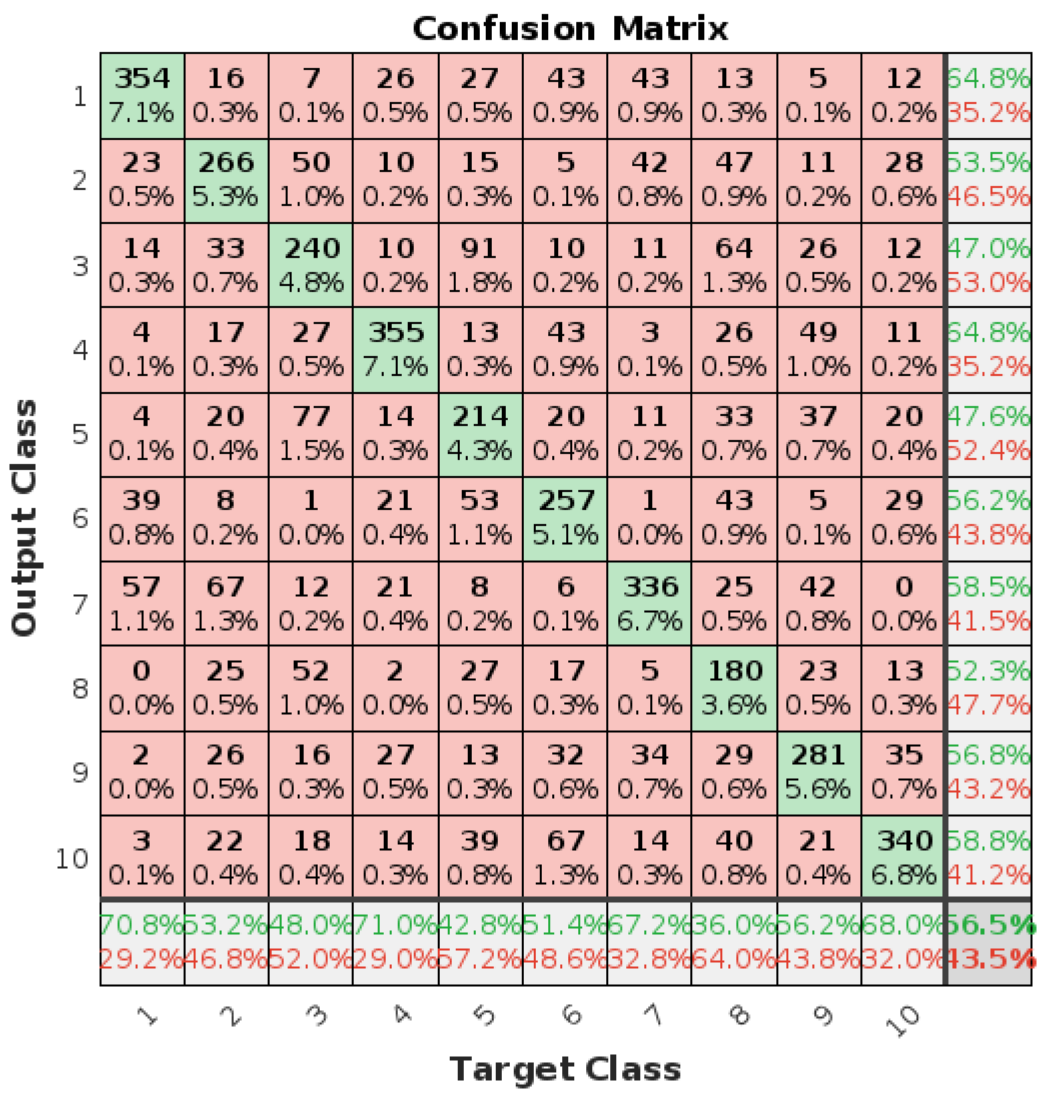
Get the number of pixels in each image

Now that the entire network is assembled, you can evaluate its performance on the test set. However, before utilizing the images with the stacked network, it's necessary to reshape the test images into a matrix format. This can be accomplished by arranging the columns of an image into a vector, and then constructing a matrix from these vectors.

```
imageWidth = 28;
imageHeight = 28;
inputSize = imageWidth*imageHeight;

% Load the test images
[xTestImages,tTest] = digitTestCellArrayData;

% Turn the test images into vectors and put them in a matrix
xTest = zeros(inputSize,numel(xTestImages));
for i = 1:numel(xTestImages)
    xTest(:,i) = xTestImages{i}(:);
end
```

You can visualize the results with a confusion matrix. The numbers in the bottom right-hand square of the matrix give the overall accuracy.

```
y = stackednet(xTest);
plotconfusion(tTest,y);
```


Confusion Matrix
Output Class
Target Class

	1	2	3	4	5	6	7	8	9	10	
1	354 7.1%	16 0.3%	7 0.1%	26 0.5%	27 0.5%	43 0.9%	43 0.9%	13 0.3%	5 0.1%	12 0.2%	64.8% 35.2%
2	23 0.5%	266 5.3%	50 1.0%	10 0.2%	15 0.3%	5 0.1%	42 0.8%	47 0.9%	11 0.2%	28 0.6%	53.5% 46.5%
3	14 0.3%	33 0.7%	240 4.8%	10 0.2%	91 1.8%	10 0.2%	11 0.2%	64 1.3%	26 0.5%	12 0.2%	47.0% 53.0%
4	4 0.1%	17 0.3%	27 0.5%	355 7.1%	13 0.3%	43 0.9%	3 0.1%	26 0.5%	49 1.0%	11 0.2%	64.8% 35.2%
5	4 0.1%	20 0.4%	77 1.5%	14 0.3%	214 4.3%	20 0.4%	11 0.2%	33 0.7%	37 0.7%	20 0.4%	47.6% 52.4%
6	39 0.8%	8 0.2%	1 0.0%	21 0.4%	53 1.1%	257 5.1%	1 0.0%	43 0.9%	5 0.1%	29 0.6%	56.2% 43.8%
7	57 1.1%	67 1.3%	12 0.2%	21 0.4%	8 0.2%	6 0.1%	336 6.7%	25 0.5%	42 0.8%	0 0.0%	58.5% 41.5%
8	0 0.0%	25 0.5%	52 1.0%	2 0.0%	27 0.5%	17 0.3%	5 0.1%	180 3.6%	23 0.5%	13 0.3%	52.3% 47.7%
9	2 0.0%	26 0.5%	16 0.3%	27 0.5%	13 0.3%	32 0.6%	34 0.7%	29 0.6%	281 5.6%	35 0.7%	56.8% 43.2%
10	3 0.1%	22 0.4%	18 0.4%	14 0.3%	39 0.8%	67 1.3%	14 0.3%	40 0.8%	21 0.4%	340 6.8%	58.8% 41.2%
	70.8% 29.2%	53.2% 46.8%	48.0% 52.0%	71.0% 29.0%	42.8% 57.2%	51.4% 48.6%	67.2% 32.8%	36.0% 64.0%	56.2% 43.8%	68.0% 32.0%	56.5% 43.5%

Step 7: Fine-Tuning the Stacked Neural Network

Fine-tuning the stacked neural network involves adjusting the parameters of the entire model, including the encoders from the autoencoders and the softmax layer, to optimize its performance on the task at hand, which in this case is digit classification. This step is crucial for refining the learned representations and improving the overall accuracy of the model. Fine-tuning typically involves using techniques like back propagation with adaptive optimization algorithms such as stochastic gradient descent (SGD) or Adam to update the weights and biases of the network. During fine-tuning, the model learns to better discriminate between different digit classes by iteratively adjusting its parameters based on the gradients of the loss function with respect to these parameters.

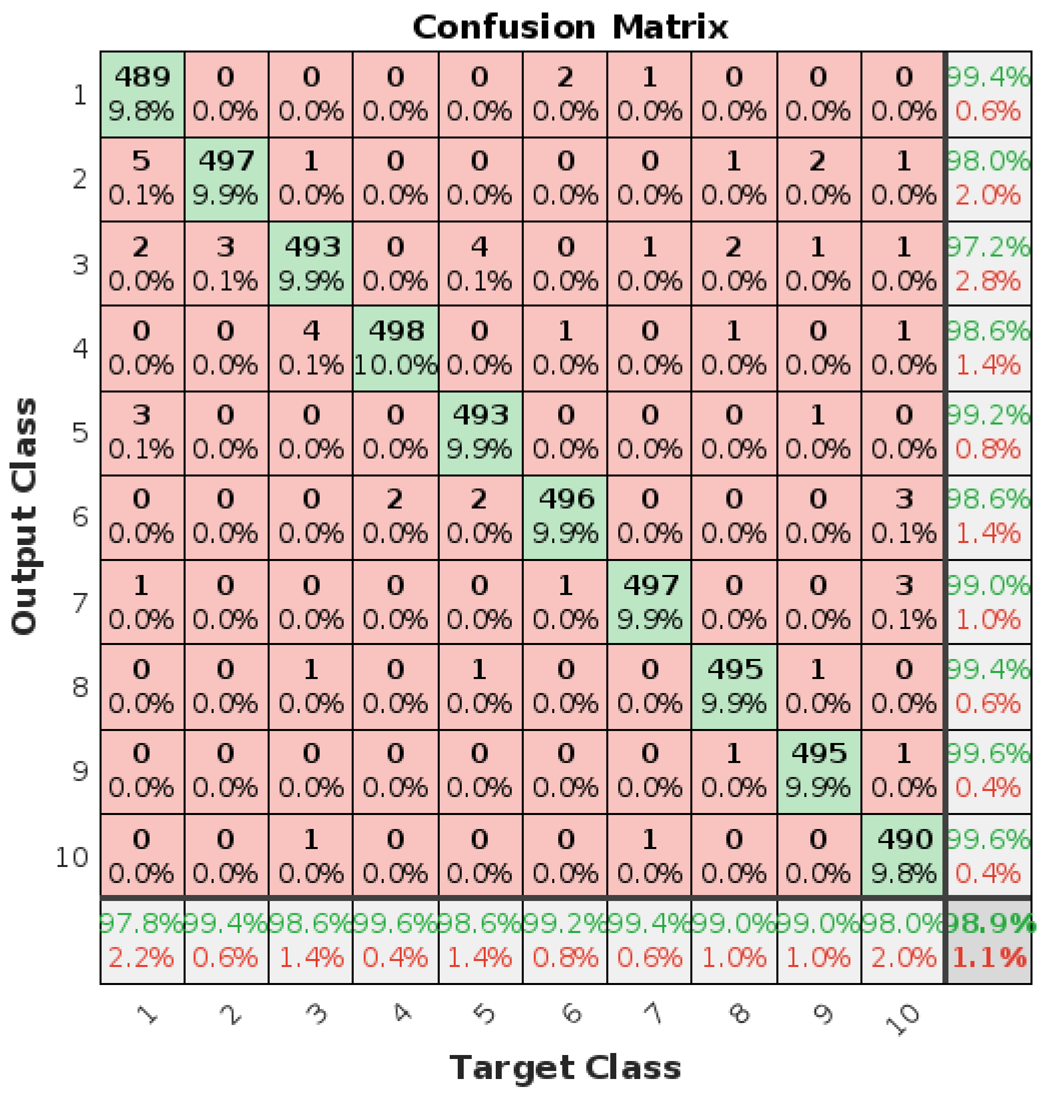
The results for the stacked neural network can be improved by performing backpropagation on the whole multilayer network. This process is often referred to as fine tuning.

You fine tune the network by retraining it on the training data in a supervised fashion. Before you can do this, you have to reshape the training images into a matrix, as was done for the test images.

```
xTrain = zeros(inputSize,numel(xTrainImages));
for i = 1:numel(xTrainImages)
    xTrain(:,i) = xTrainImages{i}(:);
end

% Perform fine tuning
stackednet = train(stackednet,xTrain,tTrain);
```

You then view the results again using a confusion matrix.

```
y = stackednet(xTest);
plotconfusion(tTest,y);
```



Confusion Matrix

Output Class \ Target Class	1	2	3	4	5	6	7	8	9	10	
1	489 9.8%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	2 0.0%	1 0.0%	0 0.0%	0 0.0%	0 0.0%	99.4% 0.6%
2	5 0.1%	497 9.9%	1 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	2 0.0%	1 0.0%	98.0% 2.0%
3	2 0.0%	3 0.1%	493 9.9%	0 0.0%	4 0.1%	0 0.0%	1 0.0%	2 0.0%	1 0.0%	1 0.0%	97.2% 2.8%
4	0 0.0%	0 0.0%	4 0.1%	498 10.0%	0 0.0%	1 0.0%	0 0.0%	1 0.0%	0 0.0%	1 0.0%	98.6% 1.4%
5	3 0.1%	0 0.0%	0 0.0%	0 0.0%	493 9.9%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	0 0.0%	99.2% 0.8%
6	0 0.0%	0 0.0%	0 0.0%	2 0.0%	2 0.0%	496 9.9%	0 0.0%	0 0.0%	0 0.0%	3 0.1%	98.6% 1.4%
7	1 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	497 9.9%	0 0.0%	0 0.0%	3 0.1%	99.0% 1.0%
8	0 0.0%	0 0.0%	1 0.0%	0 0.0%	1 0.0%	0 0.0%	0 0.0%	495 9.9%	1 0.0%	0 0.0%	99.4% 0.6%
9	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	495 9.9%	1 0.0%	99.6% 0.4%
10	0 0.0%	0 0.0%	1 0.0%	0 0.0%	0 0.0%	0 0.0%	1 0.0%	0 0.0%	0 0.0%	490 9.8%	99.6% 0.4%
	97.8% 2.2%	99.4% 0.6%	98.6% 1.4%	99.6% 0.4%	98.6% 1.4%	99.2% 0.8%	99.4% 0.6%	99.0% 1.0%	99.0% 1.0%	98.0% 2.0%	**98.9% 1.1%**

Summary

In summary, fine-tuning the stacked neural network encompasses optimizing the entire model by adjusting its parameters to enhance its ability to classify digit images accurately.

This process leverages the learned representations from the autoencoders and the classification capabilities of the softmax layer to refine the model's performance.

By iteratively updating the weights and biases of the network using backpropagation and optimization algorithms, the model learns to better capture the underlying patterns and features in the data, ultimately leading to improved classification accuracy on both training and test datasets.