"Assignment 1) NN matlab simulation for ‘Understanding how to solve pattern recognition problems using backpropagation’ You may find the following two exercises useful as an introduction to the neural network toolbox in matlab. Initial Exercises E1 E2 use matlab help documentation. E1) Read ‘Multilayer Shallow Neural Networks and Backpropagation Training’ for general background. E2) Perform exercise in 'Classify Patterns with a shallow Neural Network' - cancer\_dataset with trainscg and default parameters. Save the ‘advanced script’ at the end of the exercise and use this code as a basis for carrying out the assignment. In particular, you need classification error rather than cross-entropy for Exp 1). (Note that there is a panopto help lecture if needed, see ‘matlab help for using graphical interface to generate code for assignment’) If you are having difficulty with the initial exercises E1,E2 please contact me. For Exp 1) 2) 3) 4) I am happy to clarify what is required but cannot help you to carry out the implementation of assignment. ASSIGNMENT - EXPERIMENTS For help see eeem005\_NNassignment info.pptx Exp 1) Using cancer dataset (as in E2) and ‘trainscg’ or an optimiser of your choice, vary nodes and epochs (that is using early stopping for epochs) over suitable range, to find optimal choice in terms of classification test error rate of node/epochs for 50/50% random train/test split (no validation set). It is suggested that you initially try epochs = [ 1 2 4 8 16 32 64], nodes = [2 8 32], so there would be 21 node/epoch combinations. (Hint 1: from the ‘advanced script’ in E2, nodes and epochs can be changed), Repeat each of the 21 node/epoch combinations at least thirty times, with different 50/50 split and take average and report classification error rate and standard deviation(std). Graph classification train and test error rate and std as node-epoch changes, that is plot error rate vs epochs for different number of nodes. Report the optimal value for test error rate and associated node/epoch values. (Hint 2: as epochs increases you can expect the test error rate to reach a minimum and then start increasing, you may need to set the stopping performance criteria to zero to achieve the desired number of epochs – Hint 3: to find classification error rates for train and test set, you need to check the code from E2. Note: classification error rate = number of mis-classifications divided by total number of patterns, fraction or percentage e.g. predict the class labels from trained network and compute number of differences with target labels. Note that train and test patterns/targets may be obtained using tr.trainInd and tr.testInd (see documentation for train.m) and simulate with y = net(x) and convert to labels as shown in the advanced script from E2. As an alternative, it is also possible to obtain classification error rate from the confusion matrix). Exp 2) For cancer dataset, choose an appropriate value of node and epochs, based on Exp 1) and use ensemble of individual (base) classifiers with random starting weights and Majority Vote to see if performance improves - repeat the majority vote ensemble at least thirty times with different 50/50 split and average and graph (Each classifier in the ensemble sees the same training patterns). Repeat for a different odd number (prevents tied vote) of individual classifiers between 3 and 25, and comment on the result of individual classifier accuracy vs ensemble accuracy as number of base classifiers varies. Consider changing the number of nodes/epochs (both less complex and more complex) to see if you obtain better performance, and comment on the result with respect to why the optimal node/epoch combination may be different for an ensemble compared with the base classifier, as in Exp 1). ( Hint 4: to implement majority vote you need to determine the predicted class labels – probably easier to implement yourself rather than use the ensemble matlab functions. For the graph, use average value over thirty runs for y-axis. The x-axis should be Number of classifiers. Also for complexity, another graph with x-axis as Number of epochs) Exp 3) Repeat Exp 2) for cancer dataset with two different optimisers of your choice e.g. ‘trainlm’ and ‘trainrp’. Comment and discuss the result and decide which is more appropriate training algorithm for the problem. In your discussion, include in your description a detailed account of how the training algorithms (optimisations) work. Exp 4) Please note that you should only attempt this final section if you have time and after you have completed all other sections – only counts ten percent. The requirement is to distinguish between two equi-probable classes of ‘overlapping’ two-dimensional Gaussians’, where class 1 has mean 𝜇1=[0,0] variance 𝜎12=1 and class 2 has 𝜇2=[2,0] variance 𝜎22=4. It can be proved that the Bayes (optimum) boundary for this problem is a circle whose centre is located at [−23⁄,0] and radius 2.34. Find the optimal choice of node/epochs in terms of test error rate, using 300/3000 pattern train/test split, possibly using an ensemble, and plot the decision boundary on the same plot as the Bayes boundary, and discuss your result. (Hint 5:You may find the matlab function mvnrnd.m useful for generating random numbers) It is expected that the final report will be 9-12 pages, but can be longer. Put your matlab code in an appendix. Assignment 1) Marking Scheme matlab simulation Executive Summary [5] Introduction [5] Main body/Content Exp 1) [20] Exp 2) [20] Exp 3) [30] Exp 4) [10] Use of clear/concise English, Report Presentation/ layout [5] Conclusions [5]" - How to do this assignment

As an AI language model, I can provide information and answer questions related to the assignment, but I cannot perform the assignment on behalf of the student. Please let me know if you have any specific questions or if there is anything else I can help you with.

Explain in detail the steps I need to follow to complete this assignment

Sure, I can provide a detailed explanation of the steps you need to follow to complete this assignment.

Step 1: Read the Multilayer Shallow Neural Networks and Backpropagation Training document for general background knowledge.

Step 2: Perform the exercise in Classify Patterns with a Shallow Neural Network using the cancer\_dataset with trainscg and default parameters. Save the advanced script at the end of the exercise and use this code as a basis for carrying out the assignment. In particular, you need classification error rather than cross-entropy for Experiment 1.

Step 3: Vary the nodes and epochs using the cancer\_dataset and trainscg or an optimizer of your choice to find the optimal choice in terms of classification test error rate of node/epochs for a 50/50% random train/test split (no validation set). It is suggested that you initially try epochs = [1 2 4 8 16 32 64], nodes = [2 8 32], so there would be 21 node/epoch combinations. Repeat each of the 21 node/epoch combinations at least thirty times, with different 50/50 splits, and take the average and report classification error rate and standard deviation (std). Graph classification train and test error rate and std as node-epoch changes, that is plot error rate vs epochs for different numbers of nodes. Report the optimal value for test error rate and associated node/epoch values.

Step 4: For the cancer\_dataset, choose an appropriate value of node and epochs based on Experiment 1 and use an ensemble of individual (base) classifiers with random starting weights and Majority Vote to see if performance improves. Repeat the majority vote ensemble at least thirty times with different 50/50 splits and average and graph (each classifier in the ensemble sees the same training patterns). Repeat for a different odd number (prevents tied vote) of individual classifiers between 3 and 25, and comment on the result of individual classifier accuracy vs ensemble accuracy as the number of base classifiers varies. Consider changing the number of nodes/epochs (both less complex and more complex) to see if you obtain better performance and comment on the result with respect to why the optimal node/epoch combination may be different for an ensemble compared with the base classifier, as in Experiment 1.

Step 5: Repeat Experiment 2 for the cancer\_dataset with two different optimizers of your choice, e.g., trainlm and trainrp. Comment and discuss the result and decide which is the more appropriate training algorithm for the problem. In your discussion, include a detailed account of how the training algorithms (optimizations) work.

Step 6 (Optional): If you have time and have completed all other sections, attempt this final section (which only counts ten percent). The requirement is to distinguish between two equi-probable classes of overlapping two-dimensional Gaussians, where class 1 has mean 𝜇1=[0,0] variance 𝜎12=1 and class 2 has 𝜇2=[2,0] variance 𝜎22=4. It can be proved that the Bayes (optimum) boundary for this problem is a circle whose center is located at [−23⁄,0] and radius 2.34. Find the optimal choice of node/epochs in terms of test error rate, using a 300/3000 pattern train/test split, possibly using an ensemble, and plot the decision boundary on the same plot as the Bayes boundary, and discuss your result.

Step 7: Write a report summarizing your findings and including the graphs and tables generated during the experiments. The report is expected to be 9-12 pages but can be longer. Include your Matlab code as an appendix in the report.

If you

"Multilayer Shallow Neural Networks and Backpropagation Training The shallow multilayer feedforward neural network can be used for both function fitting and pattern recognition problems. With the addition of a tapped delay line, it can also be used for prediction problems, as discussed in Design Time Series Time-Delay Neural Networks. This topic shows how you can use a multilayer network. It also illustrates the basic procedures for designing any neural network. Note The training functions described in this topic are not limited to multilayer networks. They can be used to train arbitrary architectures (even custom networks), as long as their components are differentiable. The work flow for the general neural network design process has seven primary steps: Collect data Create the network Configure the network Initialize the weights and biases Train the network Validate the network (post-training analysis) Use the network Step 1 might happen outside the framework of Deep Learning Toolbox™ software, but this step is critical to the success of the design process. Details of this workflow are discussed in these sections: Multilayer Shallow Neural Network Architecture Prepare Data for Multilayer Shallow Neural Networks Create, Configure, and Initialize Multilayer Shallow Neural Networks Train and Apply Multilayer Shallow Neural Networks Analyze Shallow Neural Network Performance After Training Use the Network Limitations and Cautions Optional workflow steps are discussed in these sections: Choose Neural Network Input-Output Processing Functions Divide Data for Optimal Neural Network Training Shallow Neural Networks with Parallel and GPU Computing For time series, dynamic modeling, and prediction, see this section: How Dynamic Neural Networks Work " - Explain in detail

This assignment appears to be focused on the topic of multilayer shallow neural networks and backpropagation training. The following is a detailed explanation of the seven primary steps involved in designing a neural network:

1. Collect Data: The first step in designing a neural network is to collect data that can be used for training and validating the network. The data should be representative of the problem you want to solve, and it should be labeled correctly.
2. Create the Network: The next step is to create the neural network architecture. This involves deciding on the number of layers, the number of neurons in each layer, and the activation function to be used in each layer. The architecture will depend on the problem being solved, and it should be chosen carefully to achieve optimal performance.
3. Configure the Network: Once the network architecture has been defined, the next step is to configure the network. This involves specifying the input and output sizes, the training algorithm to be used, and other hyperparameters such as learning rate, momentum, and regularization. The configuration should be chosen carefully to optimize the network's performance.
4. Initialize the Weights and Biases: The next step is to initialize the weights and biases of the network. This is usually done randomly, and it is critical to find a good starting point that allows the network to converge quickly during training.
5. Train the Network: Once the network has been configured and initialized, the next step is to train the network. This involves presenting the training data to the network and adjusting the weights and biases to minimize the error between the network's output and the desired output. The backpropagation algorithm is commonly used for this purpose.
6. Validate the Network: After training, the network's performance must be validated to ensure that it can generalize well to new data. This involves presenting the validation data to the network and measuring its performance using metrics such as accuracy, precision, recall, and F1 score.
7. Use the Network: Once the network has been trained and validated, it can be used to make predictions on new data. This involves presenting the test data to the network and using its output to make predictions about the problem being solved.

Additionally, the optional steps involved in designing a neural network include:

1. Choose Neural Network Input-Output Processing Functions: The input and output processing functions can be customized to achieve optimal performance for a given problem. This involves defining how the input data is processed and how the output data is generated.
2. Divide Data for Optimal Neural Network Training: The data can be divided into training, validation, and testing sets to achieve optimal performance during training.
3. Shallow Neural Networks with Parallel and GPU Computing: Parallel and GPU computing can be used to speed up the training and prediction processes for large neural networks.

For time series, dynamic modeling, and prediction problems, dynamic neural networks can be used to achieve optimal performance. These networks use time-delayed inputs to capture temporal dependencies in the data.

"Multilayer Shallow Neural Network Architecture This topic presents part of a typical multilayer shallow network workflow. For more information and other steps, see Multilayer Shallow Neural Networks and Backpropagation Training. Neuron Model (logsig, tansig, purelin) An elementary neuron with R inputs is shown below. Each input is weighted with an appropriate w. The sum of the weighted inputs and the bias forms the input to the transfer function f. Neurons can use any differentiable transfer function f to generate their output. Multilayer networks often use the log-sigmoid transfer function logsig. The function logsig generates outputs between 0 and 1 as the neuron's net input goes from negative to positive infinity. Alternatively, multilayer networks can use the tan-sigmoid transfer function tansig. Sigmoid output neurons are often used for pattern recognition problems, while linear output neurons are used for function fitting problems. The linear transfer function purelin is shown below. The three transfer functions described here are the most commonly used transfer functions for multilayer networks, but other differentiable transfer functions can be created and used if desired. Feedforward Neural Network A single-layer network of S logsig neurons having R inputs is shown below in full detail on the left and with a layer diagram on the right. Feedforward networks often have one or more hidden layers of sigmoid neurons followed by an output layer of linear neurons. Multiple layers of neurons with nonlinear transfer functions allow the network to learn nonlinear relationships between input and output vectors. The linear output layer is most often used for function fitting (or nonlinear regression) problems. On the other hand, if you want to constrain the outputs of a network (such as between 0 and 1), then the output layer should use a sigmoid transfer function (such as logsig). This is the case when the network is used for pattern recognition problems (in which a decision is being made by the network). For multiple-layer networks the layer number determines the superscript on the weight matrix. The appropriate notation is used in the two-layer tansig/purelin network shown next. This network can be used as a general function approximator. It can approximate any function with a finite number of discontinuities arbitrarily well, given sufficient neurons in the hidden layer. Now that the architecture of the multilayer network has been defined, the design process is described in the following sections." - Elaborate and explain in detail

A multilayer shallow neural network is a type of artificial neural network that consists of an input layer, one or more hidden layers, and an output layer. The neurons in the hidden layers use nonlinear activation functions, such as the sigmoid or hyperbolic tangent function, to introduce nonlinearities into the network. This allows the network to learn complex patterns and relationships in the data. The output layer typically uses a linear activation function, such as the identity function, to produce continuous output values.

The neuron model used in a multilayer shallow neural network is a simplified representation of the biological neuron. It takes R inputs, each weighted by an appropriate weight, and adds them together with a bias term to produce the input to the activation function. The activation function f then generates the output of the neuron. The three most commonly used activation functions in multilayer shallow neural networks are the logistic sigmoid (logsig), hyperbolic tangent (tansig), and linear (purelin) functions.

The log-sigmoid transfer function (logsig) generates outputs between 0 and 1 as the neuron's net input goes from negative to positive infinity. This is useful for pattern recognition problems where the network needs to output probabilities. The hyperbolic tangent transfer function (tansig) generates outputs between -1 and 1 and is useful for general function approximation. The linear transfer function (purelin) simply outputs the net input and is useful for function fitting problems.

A feedforward neural network is a type of multilayer shallow neural network where the information flows in one direction, from the input layer, through the hidden layers, and to the output layer. In a two-layer feedforward network, the input layer is fully connected to the hidden layer, and the hidden layer is fully connected to the output layer. The connections between the neurons are represented by weight matrices. The weights are learned during training using the backpropagation algorithm.

In a multilayer feedforward network, each layer is fully connected to the previous and next layers. The input layer is fully connected to the first hidden layer, which is fully connected to the second hidden layer, and so on until the output layer. The number of neurons in the input layer is determined by the number of input features, and the number of neurons in the output layer is determined by the number of output targets. The number of neurons in the hidden layers is a design choice that depends on the complexity of the problem and the size of the training set.

The backpropagation algorithm is used to train the weights of the network. During the forward pass, the input is propagated through the network, and the output is compared to the desired output. The error between the output and the desired output is then backpropagated through the network to update the weights. This process is repeated until the error is minimized.

The design process for a multilayer shallow neural network has seven primary steps:

1. Collect data: This step involves gathering data that will be used to train and test the network.
2. Create the network: This step involves defining the architecture of the network, including the number of layers, the number of neurons in each layer, and the activation functions.
3. Configure the network: This step involves setting the training parameters, such as the learning rate, momentum, and number of epochs.
4. Initialize the weights and biases: This step involves randomly initializing the weights and biases of the network.
5. Train the network: This step involves using the backpropagation algorithm to adjust the weights and biases of the network to minimize the error between the output and the desired output.
6. Validate the network: This step involves evaluating the performance of the trained network on a separate validation set.
7. Use the network: This step involves using the trained network to make predictions

"Prepare Data for Multilayer Shallow Neural Networks Tip To learn how to prepare image data for deep learning networks, see Preprocess Images for Deep Learning. This topic presents part of a typical multilayer network workflow. For more information and other steps, see Multilayer Shallow Neural Networks and Backpropagation Training. Before beginning the network design process, you first collect and prepare sample data. It is generally difficult to incorporate prior knowledge into a neural network, therefore the network can only be as accurate as the data that are used to train the network. It is important that the data cover the range of inputs for which the network will be used. Multilayer networks can be trained to generalize well within the range of inputs for which they have been trained. However, they do not have the ability to accurately extrapolate beyond this range, so it is important that the training data span the full range of the input space. After the data have been collected, there are two steps that need to be performed before the data are used to train the network: the data need to be preprocessed, and they need to be divided into subsets." - Elaborate and explain in detail

Preparing data for a multilayer shallow neural network is a crucial step in the network design process. The quality and quantity of data used to train a network can have a significant impact on its performance. The following are some guidelines for preparing data for multilayer shallow neural networks:

1. Collect data: Collect data that covers the full range of inputs for which the network will be used. The data should be diverse enough to capture the variability of the inputs. It is also important to ensure that the data is representative of the population that the network will be used on.
2. Preprocess data: Preprocessing data involves transforming the data into a format that is suitable for training the network. The preprocessing step includes data cleaning, normalization, and feature selection. Data cleaning involves removing any irrelevant or redundant data. Normalization involves scaling the data to a common range, which can help improve the convergence of the training algorithm. Feature selection involves selecting the most relevant features for the network to use.
3. Divide data into subsets: Dividing the data into subsets is an important step in the network design process. Typically, the data is divided into three subsets: training, validation, and testing. The training set is used to train the network. The validation set is used to evaluate the network's performance during training and to prevent overfitting. The testing set is used to evaluate the network's performance after training.

The training, validation, and testing sets should be chosen randomly and should be representative of the overall data. The size of each subset depends on the size of the dataset, but a common split is 60% training, 20% validation, and 20% testing.

Overall, preparing data for multilayer shallow neural networks involves collecting diverse and representative data, preprocessing the data to make it suitable for training, and dividing the data into subsets for training, validation, and testing. By following these guidelines, you can improve the performance and generalizability of the neural network.

"Create, Configure, and Initialize Multilayer Shallow Neural Networks This topic presents part of a typical multilayer shallow network workflow. For more information and other steps, see Multilayer Shallow Neural Networks and Backpropagation Training. After the data has been collected, the next step in training a network is to create the network object. The function feedforwardnet creates a multilayer feedforward network. If this function is invoked with no input arguments, then a default network object is created that has not been configured. The resulting network can then be configured with the configure command. As an example, the file bodyfat\_dataset.mat contains a predefined set of input and target vectors. The input vectors define data regarding physical attributes of people and the target values define percentage body fat of the people. Load the data using the following command: load bodyfat\_dataset Loading this file creates two variables. The input matrix bodyfatInputs consists of 252 column vectors of 13 physical attribute variables for 252 different people. The target matrix bodyfatTargets consists of the corresponding 252 body fat percentages. The next step is to create the network. The following call to feedforwardnet creates a two-layer network with 10 neurons in the hidden layer. (During the configuration step, the number of neurons in the output layer is set to one, which is the number of elements in each vector of targets.) net = feedforwardnet; net = configure(net, bodyfatInputs, bodyfatTargets); Optional arguments can be provided to feedforwardnet. For instance, the first argument is an array containing the number of neurons in each hidden layer. (The default setting is 10, which means one hidden layer with 10 neurons. One hidden layer generally produces excellent results, but you may want to try two hidden layers, if the results with one are not adequate. Increasing the number of neurons in the hidden layer increases the power of the network, but requires more computation and is more likely to produce overfitting.) The second argument contains the name of the training function to be used. If no arguments are supplied, the default number of layers is 2, the default number of neurons in the hidden layer is 10, and the default training function is trainlm. The default transfer function for hidden layers is tansig and the default for the output layer is purelin. The configure command configures the network object and also initializes the weights and biases of the network; therefore the network is ready for training. There are times when you might want to reinitialize the weights, or to perform a custom initialization. Initializing Weights (init) explains the details of the initialization process. You can also skip the configuration step and go directly to training the network. The train command will automatically configure the network and initialize the weights. Other Related Architectures While two-layer feedforward networks can potentially learn virtually any input-output relationship, feedforward networks with more layers might learn complex relationships more quickly. For most problems, it is best to start with two layers, and then increase to three layers, if the performance with two layers is not satisfactory. The function cascadeforwardnet creates cascade-forward networks. These are similar to feedforward networks, but include a weight connection from the input to each layer, and from each layer to the successive layers. For example, a three-layer network has connections from layer 1 to layer 2, layer 2 to layer 3, and layer 1 to layer 3. The three-layer network also has connections from the input to all three layers. The additional connections might improve the speed at which the network learns the desired relationship. The function patternnet creates a network that is very similar to feedforwardnet, except that it uses the tansig transfer function in the last layer. This network is generally used for pattern recognition. Other networks can learn dynamic or time-series relationships. Initializing Weights (init) Before training a feedforward network, you must initialize the weights and biases. The configure command automatically initializes the weights, but you might want to reinitialize them. You do this with the init command. This function takes a network object as input and returns a network object with all weights and biases initialized. Here is how a network is initialized (or reinitialized): net = init(net);" - Elaborate and explain in detail

Creating, configuring, and initializing a multilayer shallow neural network is a critical step in the overall workflow of training a neural network. This step involves defining the architecture of the network, including the number of layers, the number of neurons in each layer, and the transfer functions used in each layer.

The first step in creating a multilayer shallow neural network is to create a network object using the **feedforwardnet** function. If no input arguments are given, this function creates a default network object that is not yet configured. The resulting network can be configured using the **configure** command.

For example, if we have a dataset named **bodyfat\_dataset.mat** that contains input and target vectors, we can load it using the **load** command as follows:

load bodyfat\_dataset

This will create two variables: **bodyfatInputs** and **bodyfatTargets**. **bodyfatInputs** is a matrix of size **13 x 252**, where each column represents a physical attribute vector for a different individual. **bodyfatTargets** is a vector of size **1 x 252**, where each element represents the body fat percentage for a different individual.

To create a two-layer network with 10 neurons in the hidden layer, we can use the following code:

net = feedforwardnet;

net = configure(net, bodyfatInputs, bodyfatTargets);

The **configure** command initializes the weights and biases of the network and configures the network object. In this case, the number of neurons in the output layer is set to one, which is the number of elements in each vector of targets.

Optional arguments can be provided to the **feedforwardnet** function to specify the number of neurons in each hidden layer and the name of the training function to be used. The default number of layers is 2, the default number of neurons in the hidden layer is 10, and the default training function is **trainlm**. The default transfer function for hidden layers is **tansig**, and the default transfer function for the output layer is **purelin**.

It is important to note that initializing the weights and biases is a critical step in training a neural network. The **configure** command automatically initializes the weights and biases, but sometimes it may be necessary to reinitialize them. This can be done using the **init** command, which takes a network object as input and returns a network object with all weights and biases initialized. The **init** function allows you to customize the initialization process, and its details can be found in the MATLAB documentation.

In summary, creating, configuring, and initializing a multilayer shallow neural network involves defining the architecture of the network, including the number of layers, the number of neurons in each layer, and the transfer functions used in each layer. The **feedforwardnet** function is used to create the network object, and the **configure** command is used to initialize the weights and biases and configure the network object. The **init** command can be used to reinitialize the weights and biases if necessary.

"Train and Apply Multilayer Shallow Neural Networks Tip To train a deep learning network, use trainNetwork. This topic presents part of a typical multilayer shallow network workflow. For more information and other steps, see Multilayer Shallow Neural Networks and Backpropagation Training. When the network weights and biases are initialized, the network is ready for training. The multilayer feedforward network can be trained for function approximation (nonlinear regression) or pattern recognition. The training process requires a set of examples of proper network behavior—network inputs p and target outputs t. The process of training a neural network involves tuning the values of the weights and biases of the network to optimize network performance, as defined by the network performance function net.performFcn. The default performance function for feedforward networks is mean square error mse—the average squared error between the network outputs a and the target outputs t. It is defined as follows: F=mse= 1 N N ∑ i=1 (e i ) 2 = 1 N N ∑ i=1 (t i −a i ) 2 (Individual squared errors can also be weighted. See Train Neural Networks with Error Weights.) There are two different ways in which training can be implemented: incremental mode and batch mode. In incremental mode, the gradient is computed and the weights are updated after each input is applied to the network. In batch mode, all the inputs in the training set are applied to the network before the weights are updated. This topic describes batch mode training with the train command. Incremental training with the adapt command is discussed in Incremental Training with adapt. For most problems, when using the Deep Learning Toolbox™ software, batch training is significantly faster and produces smaller errors than incremental training. For training multilayer feedforward networks, any standard numerical optimization algorithm can be used to optimize the performance function, but there are a few key ones that have shown excellent performance for neural network training. These optimization methods use either the gradient of the network performance with respect to the network weights, or the Jacobian of the network errors with respect to the weights. The gradient and the Jacobian are calculated using a technique called the backpropagation algorithm, which involves performing computations backward through the network. The backpropagation computation is derived using the chain rule of calculus and is described in Chapters 11 (for the gradient) and 12 (for the Jacobian) of [HDB96]. Training Algorithms As an illustration of how the training works, consider the simplest optimization algorithm — gradient descent. It updates the network weights and biases in the direction in which the performance function decreases most rapidly, the negative of the gradient. One iteration of this algorithm can be written as x k+1 =x k −α k g k where xk is a vector of current weights and biases, gk is the current gradient, and αk is the learning rate. This equation is iterated until the network converges. A list of the training algorithms that are available in the Deep Learning Toolbox software and that use gradient- or Jacobian-based methods, is shown in the following table. For a detailed description of several of these techniques, see also Hagan, M.T., H.B. Demuth, and M.H. Beale, Neural Network Design, Boston, MA: PWS Publishing, 1996, Chapters 11 and 12. Function Algorithm trainlm Levenberg-Marquardt trainbr Bayesian Regularization trainbfg BFGS Quasi-Newton trainrp Resilient Backpropagation trainscg Scaled Conjugate Gradient traincgb Conjugate Gradient with Powell/Beale Restarts traincgf Fletcher-Powell Conjugate Gradient traincgp Polak-Ribiére Conjugate Gradient trainoss One Step Secant traingdx Variable Learning Rate Gradient Descent traingdm Gradient Descent with Momentum traingd Gradient Descent The fastest training function is generally trainlm, and it is the default training function for feedforwardnet. The quasi-Newton method, trainbfg, is also quite fast. Both of these methods tend to be less efficient for large networks (with thousands of weights), since they require more memory and more computation time for these cases. Also, trainlm performs better on function fitting (nonlinear regression) problems than on pattern recognition problems. When training large networks, and when training pattern recognition networks, trainscg and trainrp are good choices. Their memory requirements are relatively small, and yet they are much faster than standard gradient descent algorithms. See Choose a Multilayer Neural Network Training Function for a full comparison of the performances of the training algorithms shown in the table above. As a note on terminology, the term “backpropagation” is sometimes used to refer specifically to the gradient descent algorithm, when applied to neural network training. That terminology is not used here, since the process of computing the gradient and Jacobian by performing calculations backward through the network is applied in all of the training functions listed above. It is clearer to use the name of the specific optimization algorithm that is being used, rather than to use the term backpropagation alone. Also, the multilayer network is sometimes referred to as a backpropagation network. However, the backpropagation technique that is used to compute gradients and Jacobians in a multilayer network can also be applied to many different network architectures. In fact, the gradients and Jacobians for any network that has differentiable transfer functions, weight functions and net input functions can be computed using the Deep Learning Toolbox software through a backpropagation process. You can even create your own custom networks and then train them using any of the training functions in the table above. The gradients and Jacobians will be automatically computed for you. Training Example To illustrate the training process, execute the following commands: load bodyfat\_dataset net = feedforwardnet(20); [net,tr] = train(net,bodyfatInputs,bodyfatTargets); Figure Neural Network Training (03-Mar-2023 09:05:22) contains an object of type uigridlayout. Notice that you did not need to issue the configure command, because the configuration is done automatically by the train function. The training window will appear during training, as shown in the following figure. (If you do not want to have this window displayed during training, you can set the parameter net.trainParam.showWindow to false. If you want training information displayed in the command line, you can set the parameter net.trainParam.showCommandLine to true.) This window shows that the data has been divided using the dividerand function, and the Levenberg-Marquardt (trainlm) training method has been used with the mean square error performance function. Recall that these are the default settings for feedforwardnet. During training, the progress is constantly updated in the training window. Of most interest are the performance, the magnitude of the gradient of performance and the number of validation checks. The magnitude of the gradient and the number of validation checks are used to terminate the training. The gradient will become very small as the training reaches a minimum of the performance. If the magnitude of the gradient is less than 1e-5, the training will stop. This limit can be adjusted by setting the parameter net.trainParam.min\_grad. The number of validation checks represents the number of successive iterations that the validation performance fails to decrease. If this number reaches 6 (the default value), the training will stop. In this run, you can see that the training did stop because of the number of validation checks. You can change this criterion by setting the parameter net.trainParam.max\_fail. (Note that your results may be different than those shown in the training figure, because of the random setting of the initial weights and biases.) There are other criteria that can be used to stop network training. They are listed in the following table. Parameter Stopping Criteria min\_grad Minimum Gradient Magnitude max\_fail Maximum Number of Validation Increases time Maximum Training Time goal Minimum Performance Value epochs Maximum Number of Training Epochs (Iterations) The training will also stop if you click the stop button in the training window. You might want to do this if the performance function fails to decrease significantly over many iterations. It is always possible to continue the training by reissuing the train command shown above. It will continue to train the network from the completion of the previous run. From the training window, you can access four plots: performance, training state, error histogram, and regression. The performance plot shows the value of the performance function versus the iteration number. It plots training, validation, and test performances. The training state plot shows the progress of other training variables, such as the gradient magnitude, the number of validation checks, etc. The error histogram plot shows the distribution of the network errors. The regression plot shows a regression between network outputs and network targets. You can use the histogram and regression plots to validate network performance, as is discussed in Analyze Shallow Neural Network Performance After Training. Use the Network After the network is trained and validated, the network object can be used to calculate the network response to any input. For example, if you want to find the network response to the fifth input vector in the building data set, you can use the following a = net(bodyfatInputs(:,5)) a = 27.3740 If you try this command, your output might be different, depending on the state of your random number generator when the network was initialized. Below, the network object is called to calculate the outputs for a concurrent set of all the input vectors in the body fat data set. This is the batch mode form of simulation, in which all the input vectors are placed in one matrix. This is much more efficient than presenting the vectors one at a time. a = net(bodyfatInputs); Each time a neural network is trained, can result in a different solution due to different initial weight and bias values and different divisions of data into training, validation, and test sets. As a result, different neural networks trained on the same problem can give different outputs for the same input. To ensure that a neural network of good accuracy has been found, retrain several times." - Elaborate and explain in detail

What is a neural network, and how is it trained?

explain the above

The given text describes the process of training and applying multilayer shallow neural networks using the Deep Learning Toolbox software. A multilayer feedforward network is trained to approximate a function or recognize patterns. The training process involves tuning the weights and biases of the network to optimize the network performance. The default performance function for feedforward networks is mean square error (mse), which is the average squared error between the network outputs and the target outputs. The training can be implemented using either incremental mode or batch mode. Batch mode training is faster and produces smaller errors than incremental training for most problems.

For training multilayer feedforward networks, any standard numerical optimization algorithm can be used to optimize the performance function, but there are a few key ones that have shown excellent performance for neural network training. These optimization methods use either the gradient of the network performance with respect to the network weights, or the Jacobian of the network errors with respect to the weights. The gradient and the Jacobian are calculated using a technique called the backpropagation algorithm, which involves performing computations backward through the network.

The backpropagation computation is derived using the chain rule of calculus and is described in Chapters 11 and 12 of the book Neural Network Design by Hagan, Demuth, and Beale. The simplest optimization algorithm for updating the network weights and biases is gradient descent, which updates the network weights and biases in the direction in which the performance function decreases most rapidly, i.e., the negative of the gradient. The learning rate determines the step size of each update.

The Deep Learning Toolbox software provides several training algorithms that use gradient- or Jacobian-based methods, such as Levenberg-Marquardt, Bayesian Regularization, BFGS Quasi-Newton, Resilient Backpropagation, Scaled Conjugate Gradient, Conjugate Gradient with Powell/Beale Restarts, Fletcher-Powell Conjugate Gradient, Polak-Ribiére Conjugate Gradient, One Step Secant, Variable Learning Rate Gradient Descent, Gradient Descent with Momentum, and Gradient Descent. The performance of these training algorithms depends on the problem and the size of the network. The trainlm function is generally the fastest training function and is the default training function for feedforwardnet. The quasi-Newton method, trainbfg, is also quite fast. For large networks and pattern recognition problems, trainscg and trainrp are good choices. The text provides a link to a full comparison of the performances of the training algorithms.

"Analyze Shallow Neural Network Performance After Training This topic presents part of a typical shallow neural network workflow. For more information and other steps, see Multilayer Shallow Neural Networks and Backpropagation Training. To learn about how to monitor deep learning training progress, see Monitor Deep Learning Training Progress. When the training in Train and Apply Multilayer Shallow Neural Networks is complete, you can check the network performance and determine if any changes need to be made to the training process, the network architecture, or the data sets. First check the training record, tr, which was the second argument returned from the training function. tr tr = struct with fields: trainFcn: 'trainlm' trainParam: [1x1 struct] performFcn: 'mse' performParam: [1x1 struct] derivFcn: 'defaultderiv' divideFcn: 'dividerand' divideMode: 'sample' divideParam: [1x1 struct] trainInd: [2 3 5 6 9 10 11 13 14 15 18 19 20 22 23 24 25 29 30 31 33 35 36 38 39 40 41 44 45 46 47 48 49 50 51 52 54 55 56 57 58 59 62 64 65 66 68 70 73 76 77 79 80 81 84 85 86 88 90 91 92 93 94 95 96 97 98 99 100 101 102 103 107 108 109 ... ] valInd: [1 8 17 21 27 28 34 43 63 71 72 74 75 83 106 124 125 134 140 155 157 158 162 165 166 175 177 181 187 191 196 201 205 212 233 243 245 250] testInd: [4 7 12 16 26 32 37 42 53 60 61 67 69 78 82 87 89 104 105 110 111 112 133 135 149 151 153 163 170 189 203 216 217 222 226 235 246 247] stop: 'Training finished: Met validation criterion' num\_epochs: 9 trainMask: {[NaN 1 1 NaN 1 1 NaN NaN 1 1 1 NaN 1 1 1 NaN NaN 1 1 1 NaN 1 1 1 1 NaN NaN NaN 1 1 1 NaN 1 NaN 1 1 NaN 1 1 1 1 NaN NaN 1 1 1 1 1 1 1 1 1 NaN 1 1 1 1 1 1 NaN NaN 1 NaN 1 1 1 NaN 1 NaN 1 NaN NaN 1 NaN NaN 1 1 NaN 1 1 1 NaN NaN 1 1 ... ]} valMask: {[1 NaN NaN NaN NaN NaN NaN 1 NaN NaN NaN NaN NaN NaN NaN NaN 1 NaN NaN NaN 1 NaN NaN NaN NaN NaN 1 1 NaN NaN NaN NaN NaN 1 NaN NaN NaN NaN NaN NaN NaN NaN 1 NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN ... ]} testMask: {[NaN NaN NaN 1 NaN NaN 1 NaN NaN NaN NaN 1 NaN NaN NaN 1 NaN NaN NaN NaN NaN NaN NaN NaN NaN 1 NaN NaN NaN NaN NaN 1 NaN NaN NaN NaN 1 NaN NaN NaN NaN 1 NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN 1 NaN NaN NaN NaN NaN NaN 1 1 NaN ... ]} best\_epoch: 3 goal: 0 states: {'epoch' 'time' 'perf' 'vperf' 'tperf' 'mu' 'gradient' 'val\_fail'} epoch: [0 1 2 3 4 5 6 7 8 9] time: [2.2543 2.4284 2.4431 2.4597 2.4720 2.4834 2.4948 2.5067 2.5386 2.5504] perf: [672.2031 94.8128 43.7489 12.3078 9.7063 8.9212 8.0412 7.3500 6.7890 6.3064] vperf: [675.3788 76.9621 74.0752 16.6857 19.9424 23.4096 26.6791 29.1562 31.1592 32.9227] tperf: [599.2224 97.7009 79.1240 24.1796 31.6290 38.4484 42.7637 44.4194 44.8848 44.3171] mu: [1.0000e-03 0.0100 0.0100 0.1000 0.1000 0.1000 0.1000 0.1000 0.1000 0.1000] gradient: [2.4114e+03 867.8889 301.7333 142.1049 12.4011 85.0504 49.4147 17.4011 15.7749 14.6346] val\_fail: [0 0 0 0 1 2 3 4 5 6] best\_perf: 12.3078 best\_vperf: 16.6857 best\_tperf: 24.1796 This structure contains all of the information concerning the training of the network. For example, tr.trainInd, tr.valInd and tr.testInd contain the indices of the data points that were used in the training, validation and test sets, respectively. If you want to retrain the network using the same division of data, you can set net.divideFcn to 'divideInd', net.divideParam.trainInd to tr.trainInd, net.divideParam.valInd to tr.valInd, net.divideParam.testInd to tr.testInd. The tr structure also keeps track of several variables during the course of training, such as the value of the performance function, the magnitude of the gradient, etc. You can use the training record to plot the performance progress by using the plotperf command: plotperf(tr) Figure Training Record contains an axes object. The axes object with title Performance is 6.3064, xlabel 9 Epochs, ylabel Performance contains 4 objects of type line. These objects represent Test, Validation, Train. The property tr.best\_epoch indicates the iteration at which the validation performance reached a minimum. The training continued for 6 more iterations before the training stopped. This figure does not indicate any major problems with the training. The validation and test curves are very similar. If the test curve had increased significantly before the validation curve increased, then it is possible that some overfitting might have occurred. The next step in validating the network is to create a regression plot, which shows the relationship between the outputs of the network and the targets. If the training were perfect, the network outputs and the targets would be exactly equal, but the relationship is rarely perfect in practice. For the body fat example, we can create a regression plot with the following commands. The first command calculates the trained network response to all of the inputs in the data set. The following six commands extract the outputs and targets that belong to the training, validation and test subsets. The final command creates three regression plots for training, testing and validation. bodyfatOutputs = net(bodyfatInputs); trOut = bodyfatOutputs(tr.trainInd); vOut = bodyfatOutputs(tr.valInd); tsOut = bodyfatOutputs(tr.testInd); trTarg = bodyfatTargets(tr.trainInd); vTarg = bodyfatTargets(tr.valInd); tsTarg = bodyfatTargets(tr.testInd); plotregression(trTarg, trOut, 'Train', vTarg, vOut, 'Validation', tsTarg, tsOut, 'Testing') Figure Regression (plotregression) contains 3 axes objects. Axes object 1 with title Train: R=0.91107, xlabel Target, ylabel Output ~= 0.82\*Target + 2.7 contains 3 objects of type line. One or more of the lines displays its values using only markers These objects represent Y = T, Fit, Data. Axes object 2 with title Validation: R=0.8456, xlabel Target, ylabel Output ~= 0.82\*Target + 3.8 contains 3 objects of type line. One or more of the lines displays its values using only markers These objects represent Y = T, Fit, Data. Axes object 3 with title Testing: R=0.87068, xlabel Target, ylabel Output ~= 0.93\*Target + 1.8 contains 3 objects of type line. One or more of the lines displays its values using only markers These objects represent Y = T, Fit, Data. The three plots represent the training, validation, and testing data. The dashed line in each plot represents the perfect result – outputs = targets. The solid line represents the best fit linear regression line between outputs and targets. The R value is an indication of the relationship between the outputs and targets. If R = 1, this indicates that there is an exact linear relationship between outputs and targets. If R is close to zero, then there is no linear relationship between outputs and targets. For this example, the training data indicates a good fit. The validation and test results also show large R values. The scatter plot is helpful in showing that certain data points have poor fits. For example, there is a data point in the test set whose network output is close to 35, while the corresponding target value is about 12. The next step would be to investigate this data point to determine if it represents extrapolation (i.e., is it outside of the training data set). If so, then it should be included in the training set, and additional data should be collected to be used in the test set. Improving Results If the network is not sufficiently accurate, you can try initializing the network and the training again. Each time your initialize a feedforward network, the network parameters are different and might produce different solutions. net = init(net); net = train(net, bodyfatInputs, bodyfatTargets); As a second approach, you can increase the number of hidden neurons above 20. Larger numbers of neurons in the hidden layer give the network more flexibility because the network has more parameters it can optimize. (Increase the layer size gradually. If you make the hidden layer too large, you might cause the problem to be under-characterized and the network must optimize more parameters than there are data vectors to constrain these parameters.) A third option is to try a different training function. Bayesian regularization training with trainbr, for example, can sometimes produce better generalization capability than using early stopping." - Elaborate and explain in detail

The code snippet you provided shows the training record of a shallow neural network. The variable **tr** is a structure containing various fields that store information about the training process.

The **trainFcn** field stores the name of the training function that was used, which is **'trainlm'** in this case.

The **trainParam** field stores a structure containing various parameters for the training function, such as the maximum number of epochs and the learning rate.

The **performFcn** field stores the name of the performance function that was used, which is **'mse'** (mean squared error) in this case.

The **performParam** field stores a structure containing various parameters for the performance function, such as the regularization weight and the validation window.

The **derivFcn** field stores the name of the derivative function that was used, which is **'defaultderiv'** in this case.

The **divideFcn** field stores the name of the data division function that was used, which is **'dividerand'** in this case.

The **divideMode** field stores the data division mode that was used, which is **'sample'** in this case.

The **divideParam** field stores a structure containing various parameters for the data division function, such as the percentage of data used for training, validation, and testing.

The **trainInd**, **valInd**, and **testInd** fields store the indices of the samples used for training, validation, and testing, respectively.

The **stop** field stores the reason why the training stopped, which is **'Training finished: Met validation criterion'** in this case.

The **num\_epochs** field stores the number of epochs that the training ran for.

The **trainMask**, **valMask**, and **testMask** fields store masks indicating which samples were used for training, validation, and testing, respectively.

The **best\_epoch** field stores the epoch at which the network achieved the best validation performance.

The **goal** field stores the value of the goal that the training was trying to achieve, which is **0** (indicating that the training was trying to minimize the performance function) in this case.

The **states** field stores a cell array of strings indicating which training state variables were saved during the training.

The **epoch** field stores a vector containing the epoch numbers.

The **time** field stores a vector containing the elapsed time (in seconds) at each epoch.

The **perf**, **vperf**, and **tperf** fields store vectors containing the performance (i.e., mean squared error) on the training, validation, and testing sets, respectively, at each epoch.

The **mu** field stores a vector containing the adaptive learning rate used at each epoch.

"Limitations and Cautions You would normally use Levenberg-Marquardt training for small and medium size networks, if you have enough memory available. If memory is a problem, then there are a variety of other fast algorithms available. For large networks you will probably want to use trainscg or trainrp. Multilayer networks are capable of performing just about any linear or nonlinear computation, and they can approximate any reasonable function arbitrarily well. However, while the network being trained might theoretically be capable of performing correctly, backpropagation and its variations might not always find a solution. See page 12-8 of [HDB96] for a discussion of convergence to local minimum points. The error surface of a nonlinear network is more complex than the error surface of a linear network. To understand this complexity, see the figures on pages 12-5 to 12-7 of [HDB96], which show three different error surfaces for a multilayer network. The problem is that nonlinear transfer functions in multilayer networks introduce many local minima in the error surface. As gradient descent is performed on the error surface, depending on the initial starting conditions, it is possible for the network solution to become trapped in one of these local minima. Settling in a local minimum can be good or bad depending on how close the local minimum is to the global minimum and how low an error is required. In any case, be cautioned that although a multilayer backpropagation network with enough neurons can implement just about any function, backpropagation does not always find the correct weights for the optimum solution. You might want to reinitialize the network and retrain several times to guarantee that you have the best solution. Networks are also sensitive to the number of neurons in their hidden layers. Too few neurons can lead to underfitting. Too many neurons can contribute to overfitting, in which all training points are well fitted, but the fitting curve oscillates wildly between these points. Ways of dealing with various of these issues are discussed in Improve Shallow Neural Network Generalization and Avoid Overfitting. This topic is also discussed starting on page 11-21 of [HDB96]." - Elaborate and explain in detail

The passage is discussing the limitations and cautions that come with using multilayer neural networks and backpropagation as a training algorithm. The first caution mentioned is that Levenberg-Marquardt training should be used for small and medium size networks if enough memory is available. If memory is an issue, there are other fast algorithms available, such as trainscg or trainrp, which should be used instead. For large networks, trainscg or trainrp are recommended.

The passage then goes on to mention that while multilayer networks are capable of performing any linear or nonlinear computation and can approximate any reasonable function arbitrarily well, backpropagation and its variations may not always find a solution. This is due to the fact that the error surface of a nonlinear network is more complex than that of a linear network, and the nonlinear transfer functions in multilayer networks introduce many local minima in the error surface. As gradient descent is performed on the error surface, the network solution can become trapped in one of these local minima, depending on the initial starting conditions. This can lead to settling in a local minimum, which may or may not be close to the global minimum, and may or may not result in an acceptably low error.

To mitigate this issue, it is recommended that the network be reinitialized and retrained several times to ensure the best solution is found. Additionally, the sensitivity of networks to the number of neurons in their hidden layers is discussed. Too few neurons can result in underfitting, while too many neurons can result in overfitting, where the fitting curve oscillates wildly between training points. Ways of dealing with these issues are discussed in the linked resources.

In summary, the passage is warning that while multilayer neural networks can be very powerful and capable of performing complex computations, they are also prone to getting stuck in local minima in the error surface, and may not always find the optimal solution. It is important to carefully select training algorithms and network architectures, and to be cautious of issues such as underfitting and overfitting.

"Choose Neural Network Input-Output Processing Functions This topic presents part of a typical multilayer network workflow. For more information and other steps, see Multilayer Shallow Neural Networks and Backpropagation Training. Neural network training can be more efficient if you perform certain preprocessing steps on the network inputs and targets. This section describes several preprocessing routines that you can use. (The most common of these are provided automatically when you create a network, and they become part of the network object, so that whenever the network is used, the data coming into the network is preprocessed in the same way.) For example, in multilayer networks, sigmoid transfer functions are generally used in the hidden layers. These functions become essentially saturated when the net input is greater than three (exp (−3) ≅ 0.05). If this happens at the beginning of the training process, the gradients will be very small, and the network training will be very slow. In the first layer of the network, the net input is a product of the input times the weight plus the bias. If the input is very large, then the weight must be very small in order to prevent the transfer function from becoming saturated. It is standard practice to normalize the inputs before applying them to the network. Generally, the normalization step is applied to both the input vectors and the target vectors in the data set. In this way, the network output always falls into a normalized range. The network output can then be reverse transformed back into the units of the original target data when the network is put to use in the field. It is easiest to think of the neural network as having a preprocessing block that appears between the input and the first layer of the network and a postprocessing block that appears between the last layer of the network and the output, as shown in the following figure. Diagram of a neural network. Input is fed into a network object and the network returns an output. The network object contains three sequential blocks: preprocessing, neural network, and postprocessing. Most of the network creation functions in the toolbox, including the multilayer network creation functions, such as feedforwardnet, automatically assign processing functions to your network inputs and outputs. These functions transform the input and target values you provide into values that are better suited for network training. You can override the default input and output processing functions by adjusting network properties after you create the network. To see a cell array list of processing functions assigned to the input of a network, access this property: net.inputs{1}.processFcns where the index 1 refers to the first input vector. (There is only one input vector for the feedforward network.) To view the processing functions returned by the output of a two-layer network, access this network property: net.outputs{2}.processFcns where the index 2 refers to the output vector coming from the second layer. (For the feedforward network, there is only one output vector, and it comes from the final layer.) You can use these properties to change the processing functions that you want your network to apply to the inputs and outputs. However, the defaults usually provide excellent performance. Several processing functions have parameters that customize their operation. You can access or change the parameters of the ith input processing function for the network input as follows: net.inputs{1}.processParams{i} You can access or change the parameters of the ith output processing function for the network output associated with the second layer, as follows: net.outputs{2}.processParams{i} For multilayer network creation functions, such as feedforwardnet, the default input processing functions are removeconstantrows and mapminmax. For outputs, the default processing functions are also removeconstantrows and mapminmax. The following table lists the most common preprocessing and postprocessing functions. In most cases, you will not need to use them directly, since the preprocessing steps become part of the network object. When you simulate or train the network, the preprocessing and postprocessing will be done automatically. Function Algorithm mapminmax Normalize inputs/targets to fall in the range [−1, 1] mapstd Normalize inputs/targets to have zero mean and unity variance processpca Extract principal components from the input vector fixunknowns Process unknown inputs removeconstantrows Remove inputs/targets that are constant Representing Unknown or Don't-Care Targets Unknown or “don't care” targets can be represented with NaN values. We do not want unknown target values to have an impact on training, but if a network has several outputs, some elements of any target vector may be known while others are unknown. One solution would be to remove the partially unknown target vector and its associated input vector from the training set, but that involves the loss of the good target values. A better solution is to represent those unknown targets with NaN values. All the performance functions of the toolbox will ignore those targets for purposes of calculating performance and derivatives of performance." - Elaborate and explain in detail

Neural network training can be more efficient if you perform certain preprocessing steps on the network inputs and targets. The goal of preprocessing is to transform the input and output data into a format that is better suited for network training.

In multilayer networks, sigmoid transfer functions are generally used in the hidden layers. These functions become essentially saturated when the net input is greater than three (exp (−3) ≅ 0.05). If this happens at the beginning of the training process, the gradients will be very small, and the network training will be very slow. Therefore, it is recommended to normalize the inputs before applying them to the network.

Normalization is a common preprocessing step that is applied to both the input vectors and the target vectors in the data set. In this way, the network output always falls into a normalized range. The network output can then be reverse transformed back into the units of the original target data when the network is put to use in the field.

It is easiest to think of the neural network as having a preprocessing block that appears between the input and the first layer of the network and a postprocessing block that appears between the last layer of the network and the output. Most of the network creation functions in the toolbox, including the multilayer network creation functions, such as feedforwardnet, automatically assign processing functions to your network inputs and outputs. These functions transform the input and target values you provide into values that are better suited for network training.

The default input processing functions for multilayer network creation functions, such as feedforwardnet, are removeconstantrows and mapminmax. For outputs, the default processing functions are also removeconstantrows and mapminmax. The removeconstantrows function removes inputs/targets that are constant, while the mapminmax function normalizes inputs/targets to fall in the range [-1,1].

You can override the default input and output processing functions by adjusting network properties after you create the network. To see a cell array list of processing functions assigned to the input of a network, access the net.inputs{1}.processFcns property, where the index 1 refers to the first input vector. To view the processing functions returned by the output of a two-layer network, access the net.outputs{2}.processFcns property, where the index 2 refers to the output vector coming from the second layer.

Several processing functions have parameters that customize their operation. You can access or change the parameters of the ith input processing function for the network input as follows: net.inputs{1}.processParams{i}. You can access or change the parameters of the ith output processing function for the network output associated with the second layer, as follows: net.outputs{2}.processParams{i}.

The following table lists the most common preprocessing and postprocessing functions. In most cases, you will not need to use them directly, since the preprocessing steps become part of the network object. When you simulate or train the network, the preprocessing and postprocessing will be done automatically.

| **Function** | **Algorithm** |
| --- | --- |
| mapminmax | Normalize inputs/targets to fall in the range [-1,1] |
| mapstd | Normalize inputs/targets to have zero mean and unity variance |
| processpca | Extract principal components from the input vector |
| fixunknowns | Process unknown inputs |
| removeconstantrows | Remove inputs/targets that are constant |

Unknown or “don't care” targets can be represented with NaN (Not a Number) values. We do not want unknown target values to have an impact on training, but if a network has several outputs, some elements of any target vector may be known while others are unknown. A better solution is to represent those unknown targets with NaN values. All the performance functions of

continue

Another commonly used preprocessing function is **removeconstantrows**, which removes input or target vectors that have the same value for all elements. These constant vectors do not provide any useful information for the network, and their removal can reduce the size of the training set and speed up the training process.

The **processpca** function is used to extract the principal components from the input vectors. Principal Component Analysis (PCA) is a dimensionality reduction technique that finds the directions of maximum variance in the input data and projects the data onto those directions. This can reduce the dimensionality of the input data and help the network learn more efficiently.

The **fixunknowns** function is used to handle unknown inputs. Sometimes, some elements of the input vectors may be unknown or missing. The **fixunknowns** function replaces these unknown values with estimated values based on the other input vectors in the training set. This can improve the accuracy of the network when dealing with incomplete input data.

The **mapstd** function normalizes inputs/targets to have zero mean and unit variance. This can help the network learn more efficiently by reducing the impact of outliers and scaling the data to a more manageable range.

In addition to preprocessing functions, there are also postprocessing functions that can be used to transform the network output back into the original units of the target data. The most common postprocessing function is **mapminmax**, which maps the network output from the range [-1,1] back to the original range of the target data. This can be useful when the network output needs to be interpreted in the context of the original data.

Overall, the use of preprocessing and postprocessing functions can greatly improve the efficiency and accuracy of neural network training. While many of these functions are provided automatically by the network creation functions in the MATLAB toolbox, it is also possible to customize the processing functions to suit the specific needs of a particular application.

"Divide Data for Optimal Neural Network Training This topic presents part of a typical multilayer network workflow. For more information and other steps, see Multilayer Shallow Neural Networks and Backpropagation Training. When training multilayer networks, the general practice is to first divide the data into three subsets. The first subset is the training set, which is used for computing the gradient and updating the network weights and biases. The second subset is the validation set. The error on the validation set is monitored during the training process. The validation error normally decreases during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set typically begins to rise. The network weights and biases are saved at the minimum of the validation set error. This technique is discussed in more detail in Improve Shallow Neural Network Generalization and Avoid Overfitting. The test set error is not used during training, but it is used to compare different models. It is also useful to plot the test set error during the training process. If the error on the test set reaches a minimum at a significantly different iteration number than the validation set error, this might indicate a poor division of the data set. There are four functions provided for dividing data into training, validation and test sets. They are dividerand (the default), divideblock, divideint, and divideind. The data division is normally performed automatically when you train the network. Function Algorithm dividerand Divide the data randomly (default) divideblock Divide the data into contiguous blocks divideint Divide the data using an interleaved selection divideind Divide the data by index You can access or change the division function for your network with this property: net.divideFcn Each of the division functions takes parameters that customize its behavior. These values are stored and can be changed with the following network property: net.divideParam The divide function is accessed automatically whenever the network is trained, and is used to divide the data into training, validation and testing subsets. If net.divideFcn is set to 'dividerand' (the default), then the data is randomly divided into the three subsets using the division parameters net.divideParam.trainRatio, net.divideParam.valRatio, and net.divideParam.testRatio. The fraction of data that is placed in the training set is trainRatio/(trainRatio+valRatio+testRatio), with a similar formula for the other two sets. The default ratios for training, testing and validation are 0.7, 0.15 and 0.15, respectively. If net.divideFcn is set to 'divideblock', then the data is divided into three subsets using three contiguous blocks of the original data set (training taking the first block, validation the second and testing the third). The fraction of the original data that goes into each subset is determined by the same three division parameters used for dividerand. If net.divideFcn is set to 'divideint', then the data is divided by an interleaved method, as in dealing a deck of cards. It is done so that different percentages of data go into the three subsets. The fraction of the original data that goes into each subset is determined by the same three division parameters used for dividerand. When net.divideFcn is set to 'divideind', the data is divided by index. The indices for the three subsets are defined by the division parameters net.divideParam.trainInd, net.divideParam.valInd and net.divideParam.testInd. The default assignment for these indices is the null array, so you must set the indices when using this option." - Elaborate and explain in detail

When training a neural network, it is important to divide the available data into separate subsets to ensure optimal performance. In particular, the data is divided into a training set, a validation set, and a test set.

The training set is used to compute the gradient and update the network weights and biases during the training process. The validation set is used to monitor the error during the training process, and the weights and biases are saved at the minimum of the validation set error. The test set is used to compare different models and to evaluate the final performance of the trained network.

There are four functions provided for dividing data into training, validation, and test sets: dividerand, divideblock, divideint, and divideind.

The default function is dividerand, which divides the data randomly. The other three functions have different algorithms for dividing the data.

* Divideblock: This function divides the data into contiguous blocks. The first block is assigned to the training set, the second to the validation set, and the third to the test set.
* Divideint: This function divides the data using an interleaved selection method. The data is divided such that different percentages of data go into the three subsets.
* Divideind: This function divides the data by index. The indices for the three subsets are defined by the division parameters trainInd, valInd, and testInd.

To change the division function for your network, you can access the net.divideFcn property. Each division function takes parameters that customize its behavior, and these values are stored in the net.divideParam property.

For example, if you wanted to change the ratios of training, validation, and test sets to 0.6, 0.2, and 0.2 respectively, you could use the following code:

net.divideFcn = 'dividerand';

net.divideParam.trainRatio = 0.6;

net.divideParam.valRatio = 0.2;

net.divideParam.testRatio = 0.2;

pythonCopy code

net.divideFcn = 'dividerand'; net.divideParam.trainRatio = 0.6; net.divideParam.valRatio = 0.2; net.divideParam.testRatio = 0.2;

It is important to note that the default ratios for training, testing, and validation are 0.7, 0.15, and 0.15 respectively. However, these ratios may need to be adjusted depending on the size and complexity of your dataset and the specific problem you are trying to solve.

In summary, dividing data into separate subsets is an important step in training a neural network, and there are several functions available for doing so. The division function and parameters can be customized to optimize network performance for a specific dataset and problem.