1. Why is it generally preferable to use a Logistic Regression classifier rather than a classical Perceptron (i.e., a single layer of linear threshold units trained using the Perceptron training algorithm)? How can you tweak a Perceptron to make it equivalent to a Logistic Regression classifier?

A1. Logistic Regression and classical Perceptron are two linear models for binary classification. The main difference between them is that Logistic Regression outputs the probability of the positive class, while Perceptron directly outputs the class label. This means that Logistic Regression can be more easily extended to multiclass classification and can also estimate the uncertainty of its predictions.

Another important difference is the loss function used during training. The Perceptron algorithm minimizes the number of misclassifications, which can lead to multiple solutions. On the other hand, Logistic Regression minimizes the negative log-likelihood of the data under the assumption of a Bernoulli distribution, which has a unique solution.

To make a Perceptron equivalent to a Logistic Regression classifier, we need to modify the activation function used by the Perceptron to the logistic sigmoid function. This can be achieved by replacing the step function used to compute the output of the Perceptron with the sigmoid function. We can then train the Perceptron using the same gradient descent algorithm used for Logistic Regression, but with the modified activation function.

1. Why was the logistic activation function a key ingredient in training the first MLPs?

A2. The logistic activation function, also known as the sigmoid function, was a key ingredient in training the first Multilayer Perceptrons (MLPs) because it provides a smooth, differentiable, and nonlinear output. The logistic function maps any input value to a value between 0 and 1, which can be interpreted as a probability. This makes it a good choice for binary classification problems, where the output can be interpreted as the probability of belonging to a certain class.

The logistic function's derivative has a simple expression in terms of the function itself, which makes it easy to compute the gradients needed for backpropagation during training. Additionally, the logistic function's outputs are bounded, which can help prevent numerical overflow during training. For these reasons, the logistic function was a popular choice for the activation function in early MLPs.

1. Name three popular activation functions. Can you draw them?

A3.

1. Suppose you have an MLP composed of one input layer with 10 passthrough neurons, followed by one hidden layer with 50 artificial neurons, and finally one output layer with 3 artificial neurons. All artificial neurons use the ReLU activation function.
   * What is the shape of the input matrix **X**?
   * What about the shape of the hidden layer’s weight vector **W***h*, and the shape of its bias vector **b***h*?
   * What is the shape of the output layer’s weight vector **W***o*, and its bias vector **b***o*?
   * What is the shape of the network’s output matrix **Y**?
   * Write the equation that computes the network’s output matrix **Y** as a function of **X**, **W***h*, **b***h*, **W***o* and **b***o*.

A4.

• The shape of the input matrix X is (batch\_size, 10).

• The shape of the hidden layer’s weight vector Wh is (10, 50), and the shape of its bias vector bh is (50,).

• The shape of the output layer’s weight vector Wo is (50, 3), and its bias vector bo is (3,).

• The shape of the network’s output matrix Y is (batch\_size, 3).

• The equation that computes the network’s output matrix Y as a function of X, Wh, bh, Wo, and bo is:

Y = ReLU(ReLU(X @ Wh + bh) @ Wo + bo)

where @ represents the matrix multiplication operation and ReLU(x) applies the ReLU activation function elementwise to the tensor x.

1. How many neurons do you need in the output layer if you want to classify email into spam or ham? What activation function should you use in the output layer? If instead you want to tackle MNIST, how many neurons do you need in the output layer, using what activation function?

A5. For classifying email into spam or ham, you need only one neuron in the output layer since it is a binary classification problem. In the output layer, it is common to use the sigmoid activation function, which returns a value between 0 and 1 indicating the predicted probability of the email being spam.

For tackling the MNIST dataset, you need 10 neurons in the output layer, since there are 10 possible classes (digits 0 through 9). The activation function used in the output layer depends on the specific task. For multiclass classification problems like MNIST, it is common to use the softmax activation function, which normalizes the outputs to probabilities that sum up to 1. This means that each output can be interpreted as the predicted probability of the corresponding class.

1. What is backpropagation and how does it work? What is the difference between backpropagation and reverse-mode autodiff?

A6. Backpropagation is a method used to train neural networks. It is based on the chain rule of calculus and allows us to compute the gradients of the loss function with respect to the weights of the network. These gradients are then used to update the weights of the network through an optimization algorithm such as stochastic gradient descent (SGD).

Backpropagation works by computing the gradient of the loss function with respect to the output of each layer in the network, starting from the output layer and moving backwards through the network. Once the gradients of the output layer are computed, they are used to compute the gradients of the previous layer, and so on, until the gradients of all layers have been computed.

Reverse-mode autodiff is a general method for computing gradients of a function with respect to its inputs. It is based on the chain rule of calculus, like backpropagation, but it can be used to compute gradients of any function, not just those that are implemented by a neural network. In reverse-mode autodiff, the function is evaluated in the forward pass, and the gradient of the function with respect to its inputs is computed in the backward pass, using the chain rule. In contrast, backpropagation is a specific algorithm for computing gradients in neural networks, and it is optimized for efficiency by taking advantage of the structure of the network.

1. Can you list all the hyperparameters you can tweak in an MLP? If the MLP overfits the training data, how could you tweak these hyperparameters to try to solve the problem?

A7.   
Here are some hyperparameters that can be tweaked in an MLP:

* Learning rate
* Number of layers
* Number of neurons per layer
* Activation function
* Regularization parameters (e.g. L1 or L2 regularization)
* Batch size
* Number of epochs
* Optimizer (e.g. SGD, Adam)

If the MLP overfits the training data, here are some ways to try to solve the problem:

* Reduce the number of neurons per layer or the number of layers to reduce the model's capacity
* Increase the regularization parameter to penalize complex models
* Use early stopping to prevent the model from overfitting by stopping training when validation loss starts increasing
* Use dropout to randomly drop out neurons during training to prevent over-reliance on specific neurons or features
* Increase the amount of training data to improve generalization

1. Train a deep MLP on the MNIST dataset and see if you can get over 98% precision. Try adding all the bells and whistles (i.e., save checkpoints, restore the last checkpoint in case of an interruption, add summaries, plot learning curves using TensorBoard, and so on).

A8. As a language model, I don't have access to train deep MLPs on the MNIST dataset, as it requires high computational resources. However, I can provide you with a general outline of how to approach this task and the steps involved.

1. Load the MNIST dataset using PyTorch's DataLoader.
2. Define the architecture of your MLP using the nn.Module class.
3. Set the hyperparameters for your model, such as learning rate, batch size, number of epochs, number of hidden layers, number of neurons in each hidden layer, dropout rate, weight initialization, etc.
4. Define the loss function and the optimizer you will use for training the model.
5. Train the model using the training dataset, and validate it using the validation dataset. Monitor the loss and accuracy during training to ensure the model is improving.
6. Once the model is trained, evaluate its performance on the test dataset to see its final accuracy.
7. Save the model's checkpoints during training, and restore the last checkpoint in case of an interruption.
8. Add summaries to monitor the model's progress during training, and plot learning curves using TensorBoard.

To improve the model's accuracy, you can try tweaking various hyperparameters such as increasing the number of hidden layers, adjusting the number of neurons in each layer, changing the learning rate, trying different weight initialization methods, experimenting with different activation functions, or adding regularization techniques like dropout or weight decay. Additionally, you can try different optimization algorithms like Adam or RMSprop. It is also important to make sure that the model is not overfitting to the training dataset by monitoring its performance on the validation set.