Deep Learning - Lab 3

1. Grayscale Image Representation: Convert the original image to grayscale. This simplifies the problem, as we're only dealing with intensity values.

Edge Detection Filter: Define an edge detection filter or kernel. A commonly used 1D edge detection filter is the Sobel filter.

Convolution Operation: Slide the 1D kernel over each row (or column) of the grayscale image. At each step, perform element-wise multiplication between the kernel and the corresponding section of the image, and then sum up the results to get a single value.

Edge Response Map: Create a new image (referred to as an "edge response map") where each pixel value corresponds to the result of the convolution operation applied at that position in the original image.

By sliding the kernel across the entire image and performing the convolution operation at each position, you'll obtain an edge response map that highlights areas where the intensity changes, which often corresponds to edges.

Thresholding: To make the edges more visible, you can apply a threshold to the edge response map. Pixels with values above a certain threshold are considered as part of an edge, while pixels with values below the threshold are considered as part of the background.

Post-Processing: Often, post-processing steps such as non-maximum suppression (to thin the edges) and edge linking (to connect edges) are applied to refine the edge detection results.

03.

1. Overfitting: Overfitting occurs when a model learns to perform very well on the training data but fails to generalize to new, unseen data (like validation data). As you train for more epochs, the model may start memorizing the training data rather than learning meaningful features. This can cause the validation error to increase because the model's ability to generalize decreases. To address this, you can introduce regularization techniques like dropout, L2 regularization, or early stopping to prevent the model from overfitting.

Learning Rate: If the learning rate is too high, the optimization process might become unstable and prevent the model from converging to a good solution. Alternatively, if the learning rate is too low, the model might converge very slowly and not reach its full potential even after many epochs. Experimenting with different learning rates can help find a balance between rapid convergence and stable training.

Learning Rate Scheduling: If you're using a fixed learning rate throughout training, it might be beneficial to implement learning rate scheduling. This involves reducing the learning rate during training (e.g., by a factor of 2) after a certain number of epochs. This can help the model fine-tune its weights more effectively as training progresses.

Data Augmentation: If you're not using data augmentation, the model might start memorizing the training data, especially if the dataset is relatively small. Data augmentation introduces variations in the training data, helping the model learn more robust features and reducing overfitting.

Model Architecture: The complexity of your model architecture might not be appropriate for the dataset. A model that's too complex can overfit the training data. You might need to adjust the number of layers, the number of neurons, or other architectural aspects to strike the right balance.

Batch Size: The batch size used during training can also affect the convergence behavior. A smaller batch size might introduce more noise into the gradient estimates, which could hinder convergence. Conversely, a very large batch size might cause the model to converge too quickly, preventing it from finding a good solution.

Random Initialization: Neural networks are often initialized with random weights. The random initialization might lead to models that get stuck in poor local minima. Trying different random seeds or using more sophisticated weight initialization methods might help.

Data Quality: Poor quality or noisy training data can negatively impact the model's ability to generalize. Ensure that your training data is clean and representative of the problem you're trying to solve.

Model Complexity: If your model is very complex and the dataset is relatively small, it might be prone to overfitting. Consider simplifying the model or using techniques like transfer learning.

Validation Split: Ensure that the validation data is representative of the test data and isn't biased in some way. If the validation set is not a good representation of the general data distribution, the observed validation error might not be indicative of the model's true performance.

1. Explain how you can modify the training process to stop that from happening.

Random Seed: Set a random seed for reproducibility. This ensures that random initialization and other stochastic aspects of training remain consistent across different runs, making it easier to diagnose issues.

Cross-Validation: Use techniques like k-fold cross-validation to evaluate your model's performance more robustly. This involves training and validating the model on different subsets of the data to get a better estimate of its generalization performance.

Model Complexity: Consider reducing the complexity of the model if it's too large for the given dataset. This might involve reducing the number of layers, the number of neurons in each layer, or even using a simpler architecture altogether.

Batch Normalization: Add batch normalization layers to normalize the outputs of intermediate layers during training. This can stabilize the learning process and reduce the likelihood of overfitting.

Validation Frequency: Increase the frequency of validation checks during training. This helps you detect overfitting sooner and apply appropriate countermeasures.

1. Explain how the mini batch SGD (Stochastic Gradient Descent) algorithm can converge faster than the batch Gradient Descent algorithm.

Batch Gradient Descent:

In Batch Gradient Descent, the entire training dataset is used to compute the gradient of the loss function with respect to the model parameters in a single step. This means that the gradient is computed over the entire dataset, and the model parameters are updated only once per epoch. While this approach provides accurate gradient estimates, it can be computationally expensive and slow, especially for large datasets. Additionally, the updates tend to be very smooth and consistent.

Mini-Batch Stochastic Gradient Descent:

Mini-Batch SGD takes a middle-ground approach between the extreme of Batch Gradient Descent and the randomness of pure Stochastic Gradient Descent. Instead of using the entire dataset or a single data point, Mini-Batch SGD uses a small subset (mini-batch) of the training data to compute the gradient and update the model parameters. The mini-batch size is typically chosen to be a value between 1 and the total dataset size. This approach combines some benefits of both Batch Gradient Descent and Stochastic Gradient Descent.

The key advantage of Mini-Batch SGD is its ability to converge faster compared to Batch Gradient Descent, and this can be attributed to a few factors:

Faster Updates: Because Mini-Batch SGD updates the model parameters more frequently (once per mini-batch), it can escape shallow local minima or saddle points more quickly. This faster update rate allows the optimization process to jump out of regions where the gradient might be shallow.

Efficient Computation: Computing the gradient over a small mini batch is more computationally efficient compared to computing it over the entire dataset. This allows for faster iterations and better utilization of hardware resources like GPUs.

Noise Introduction: The mini-batch approach introduces some randomness into the gradient estimation due to the smaller sample size. While this might seem counterintuitive, this noise can actually help the optimization process by making the updates less deterministic. It can help the model jump out of local minima and explore different directions.

Better Generalization: The noise introduced by Mini-Batch SGD acts as a form of implicit regularization. It can prevent the model from overfitting to the training data and help it generalize better to unseen data.

Adaptive Learning Rates: Many variations of Mini-Batch SGD, such as Adam and RMSProp, use adaptive learning rates that adjust based on the historical gradient information. These techniques can help the algorithm converge faster and more robustly.