Assignment- 2- Deep Learning

Kisalay Ghosh

3rd March 2025

(1) Why is L_2 regularization known as weight decay? How is it related to early stopping?

L_2 regularization, also known as Ridge Regression, adds a penalty term $\lambda\|w\|^2$ to the loss function, where w represents the model weights. This results in the following update rule during gradient descent:

$$w := w - \eta\frac{\partial L}{\partial w} - \eta\lambda w \tag{1}$$

This update shows that each weight is multiplied by $(1 - \eta\lambda)$ in every step, effectively shrinking the weight values over time. This gradual decay of weights is why L_2 regularization is often called **weight decay**.

Relation to Early Stopping:

- Early stopping is a form of implicit regularization where training is halted when validation loss stops improving.
- Like L_2 regularization, early stopping prevents large weights and reduces overfitting.
- Both methods improve generalization, but early stopping dynamically adjusts the stopping point, whereas L_2 continuously shrinks weights throughout training.

(2) Why does L_1 regularization lead to more sparsity than L_2 regularization?

L_1 regularization (Lasso Regression) adds the term $\lambda\|w\|_1$ to the loss function. Unlike L_2, which squares the weights, L_1 applies an absolute value penalty, leading to the following weight update:

$$w := w - \eta\frac{\partial L}{\partial w} - \eta\lambda \cdot \text{sign}(w) \tag{2}$$

This update subtracts a constant value (λ) from the weight at every step, which can eventually push small weights to exactly **zero**, creating sparse models.

Key Differences:

- L_2 regularization shrinks all weights smoothly but does not force them to zero.
- L_1 regularization removes some weights entirely by setting them to zero, leading to a more compact and interpretable model.

(3) Advantage of using Algorithm 7.2 or 7.3 over Algorithm 7.1

Algorithm 7.1 discusses the effects of L_1 and L_2 regularization but does not provide a direct solution for handling unstable optimization problems.

Advantages of Algorithm 7.2 or 7.3:

- **Algorithm 7.2: Explicit Constraints and Reprojection Methods**
  Instead of applying static penalties, Algorithm 7.2 constrains weight magnitudes dynamically using projection techniques. This prevents instability due to weight explosion while maintaining effective optimization.
- **Algorithm 7.3: Regularization for Underdetermined Problems**
  Algorithm 7.3 ensures convergence in ill-posed problems where models require stabilization, such as logistic regression and PCA with singular matrices. It modifies matrix inversion techniques to guarantee numerical stability, preventing divergence.
- **Better Numerical Stability**
  Unlike standard L_2 weight decay, which only shrinks weights, these algorithms provide more robust constraints that guarantee **bounded weight values** and **convergence of iterative optimization methods**.
- **Avoiding Dead Units**
  Explicit constraints in Algorithm 7.2 prevent neural networks from developing inactive units due to excessive regularization pressure.
- **Application to Linear Algebra Problems**
  Algorithm 7.3 extends beyond deep learning, stabilizing linear regression and PCA using a variation of the Moore-Penrose pseudoinverse.

Thus, Algorithms 7.2 and 7.3 offer improved **stability, convergence, and numerical robustness** compared to Algorithm 7.1.

Problem 2- (1) Pros and cons of small and large minibatch sizes in Algorithm 8.1

Minibatch stochastic gradient descent (SGD) balances the efficiency of batch gradient descent and the noisy updates of pure stochastic gradient descent. The choice of minibatch size has important consequences:

Pros and Cons of Small Minibatch Sizes:

- **Pros:**
  - Faster updates, leading to more frequent weight adjustments and quicker convergence.
  - More stochasticity, which helps in escaping local minima and saddle points.
  - Requires less memory, making it suitable for devices with limited GPU memory.
- **Cons:**
  - Noisier gradient estimates, leading to unstable updates.
  - Less efficient for parallel computing since smaller batches do not fully utilize hardware.

Pros and Cons of Large Minibatch Sizes:

- **Pros:**
  - More stable and accurate gradient estimates, leading to smoother convergence.
  - More efficient parallelization, making better use of modern GPUs.
  - Allows for larger learning rates due to lower gradient variance.
- **Cons:**
  - Requires more memory, limiting the batch size on GPU-constrained hardware.
  - Can lead to sharp minimizers and poor generalization.

(2) How the momentum term in Algorithm 8.2 helps overcome plateaus and speeds up learning

Momentum is a technique that helps smooth gradient updates by accumulating a moving average of past gradients. The update rule with momentum is:

$$v_t = \beta v_{t-1} + (1 - \beta)\nabla_\theta J(\theta) \tag{3}$$

$$\theta := \theta - \eta v_t \tag{4}$$

Advantages of Momentum:

- Helps accelerate training in regions with consistent gradients, reducing oscillations and speeding up convergence.
- Helps escape flat regions (plateaus) where gradients are small by accumulating past updates.
- Reduces zigzagging behavior in steep valleys, leading to smoother optimization.

(3) How Adam (Algorithm 8.7) improves momentum estimation over Algorithm 8.2

Adam (Adaptive Moment Estimation) combines momentum with adaptive learning rates by maintaining both first and second moment estimates of the gradient:

$$s_t = \beta_1 s_{t-1} + (1 - \beta_1)\nabla_\theta J(\theta) \tag{5}$$

$$r_t = \beta_2 r_{t-1} + (1 - \beta_2)(\nabla_\theta J(\theta))^2 \tag{6}$$

Advantages of Adam:

- Unlike momentum, Adam adapts the learning rate for each parameter individually using the second moment estimate.
- Bias correction ensures stable learning in the early iterations.
- More robust to poor hyperparameter choices compared to basic momentum-based methods.

(4) Adaptive learning rates in Algorithms 8.4 and 8.5

Algorithm 8.4 (Adaptive Learning Rate via Initialization):

- It emphasizes choosing appropriate initial weight scales and bias values to ensure stable gradient propagation.
- Poor initialization can lead to vanishing or exploding gradients, requiring learning rate adjustments.

Algorithm 8.5 (RMSProp):

- RMSProp adapts learning rates using an exponentially decaying average of squared gradients:

$$r_t = \rho r_{t-1} + (1 - \rho)(\nabla_\theta J(\theta))^2 \tag{7}$$

- This normalizes gradients dynamically, preventing extreme oscillations in the parameter space.
- Unlike AdaGrad, which shrinks learning rates indefinitely, RMSProp maintains a stable learning rate.

Problem 3- (1) Expected side effects of dropout

Dropout is a regularization technique where neurons are randomly deactivated during training. The expected side effects include:

- Increased robustness by preventing co-adaptation of neurons, leading to better generalization.
- Increased noise in training updates, which can slow down convergence.
- Reduced effective network capacity, potentially limiting expressiveness.
- Need for higher training epochs to reach optimal performance due to the stochastic nature of dropout.

(2) Differences between bagging and dropout

Dropout is often compared to bagging since both techniques introduce model diversity. The key differences are:

- **Bagging:** Trains multiple separate models and combines their predictions, typically via averaging or voting.
- **Dropout:** Trains a single model but effectively creates an ensemble by randomly dropping neurons in different forward passes.
- Dropout is applied at the neuron level, whereas bagging operates at the model level.
- Bagging requires multiple complete models, increasing storage, while dropout requires only one network.

(3) Expected side effects of batch normalization

Batch normalization (BN) normalizes activations within each minibatch, improving stability. However, it has some side effects:

- **Reduced expressiveness:** By forcing activations into a standard distribution, BN may limit a layer's ability to learn highly complex features.
- **Dependence on minibatch statistics:** If batch sizes are too small, BN estimates become unreliable, causing fluctuations during training.
- **Shift in internal representation:** BN alters the learned representations, sometimes requiring fine-tuning of hyperparameters like learning rates.

(4) Undesirable side effects of batch normalization at test time

During training, BN normalizes using minibatch statistics. However, at test time:

- Test samples are not part of a minibatch, so running estimates of mean and variance must be used instead.
- If training and test distributions differ significantly, these estimates may introduce a distribution shift.
- The performance may degrade if the model was over-reliant on the training batch statistics.

(5) How dropout and batch normalization should be used jointly

Dropout and batch normalization can be combined effectively, but with careful tuning:

- BN stabilizes activations, reducing the internal covariate shift, making dropout less necessary in some cases.
- Dropout should be applied after BN rather than before, as BN normalizing a sparse activation map may lead to instability.
- Some architectures may require reduced dropout rates when using BN, as BN already regularizes learning.

Problem 4- (1) Training a ReLU neural network and plotting training error

A fully connected neural network is designed to approximate the function:

$$g(x) = \sin\left(\frac{\pi}{4}x\right) \tag{8}$$

Network Design:

- Input: Single scalar value x.
- Hidden Layers: Fully connected layers with ReLU activations.
- Output Layer: Single neuron with a linear activation function.
- Training Data: 200 samples of x in the range $[-2, 2]$.

- Loss Function: Mean Squared Error (MSE).
- Optimizer: Adam with a learning rate of 0.01.
- Training: Run for multiple epochs until error converges.

The training error (average and maximal error) is plotted per epoch to analyze convergence.

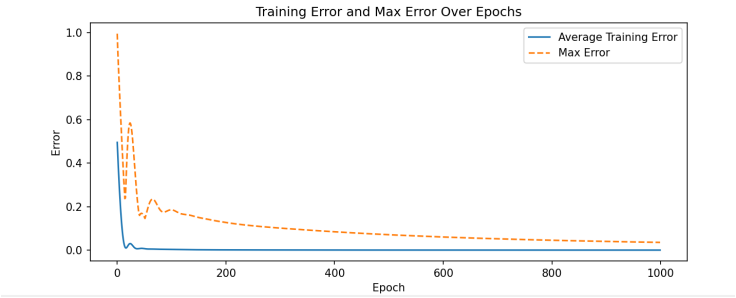

Figure 1: Training Error and Max Error Over Epochs

(2) Computing activation regions and visualizing patterns

ReLU networks introduce piecewise linear regions due to the activation behavior:

- Each neuron is either **active** (gradient=1) or **inactive** (gradient=0), forming a binary activation pattern.
- Each activation pattern defines an activation region where all inputs share the same activation state.
- We compute these activation regions and color them on the $[-2, 2]$ range.

For each region, the activation pattern (binary string) is determined and visualized using a color-coded plot.

(3) Quantifying activation pattern changes using Hamming distance

To measure network dynamics during training:

- Compute activation patterns at the start and end of each epoch.
- Compute the Hamming distance between the binary activation patterns of all training samples.

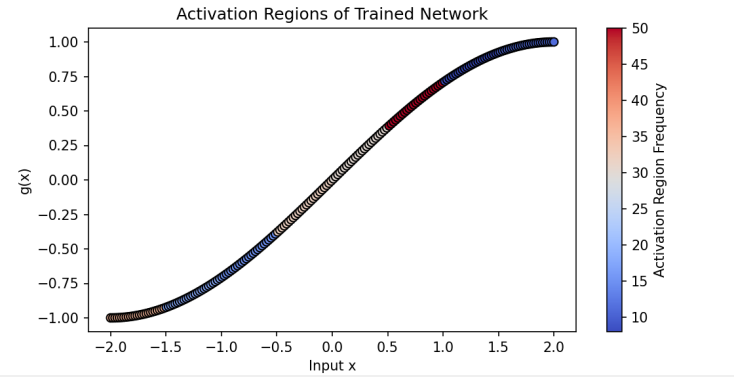

Figure 2: Activation Regions of Trained Network

- Plot the total Hamming distance per epoch to observe pattern stability.
- Verify if the activation pattern stabilizes over time.

If the Hamming distance reduces over epochs, it indicates stabilization in network behavior. The last epoch should ideally have a similar activation pattern at its start and end.

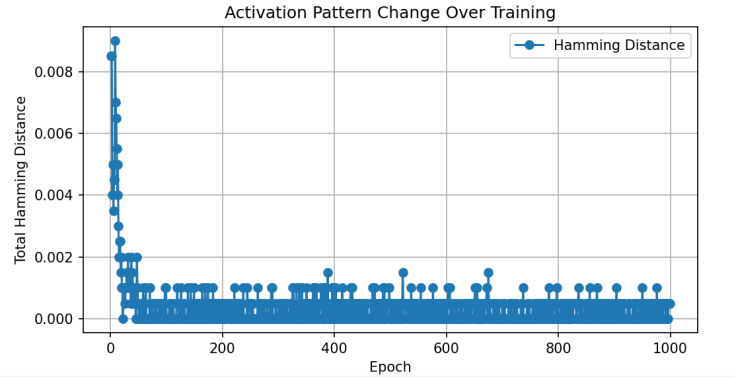

Figure 3: Activation Pattern Change Over Training

Problem 4: ReLU Neural Network Approximation

(1) Training a ReLU neural network and plotting training error

A fully connected neural network is designed to approximate the function:

$$g(x) = \sin\left(\frac{\pi}{4}x\right) \tag{9}$$

Network Design:

- Input: Single scalar value x.
- Hidden Layers: Fully connected layers with ReLU activations.
- Output Layer: Single neuron with a linear activation function.
- Training Data: 200 samples of x in the range $[-2, 2]$.
- Loss Function: Mean Squared Error (MSE).
- Optimizer: Adam with a learning rate of 0.01.
- Training: Run for multiple epochs until error converges.

The training error (average and maximal error) is plotted per epoch to analyze convergence.

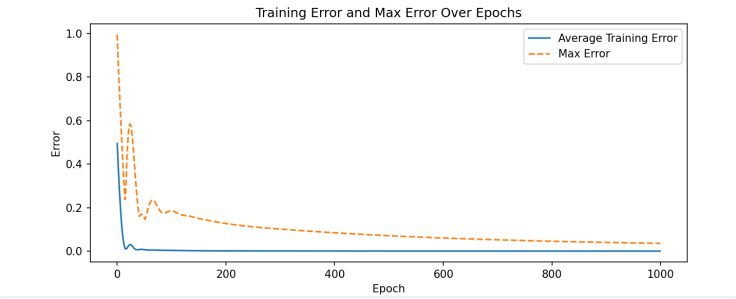

Figure 4: Training Error and Max Error Over Epochs

(2) Computing activation regions and visualizing patterns

ReLU networks introduce piecewise linear regions due to the activation behavior:

- Each neuron is either **active** (gradient=1) or **inactive** (gradient=0), forming a binary activation pattern.

- Each activation pattern defines an activation region where all inputs share the same activation state.
- We compute these activation regions and color them on the $[-2, 2]$ range.

For each region, the activation pattern (binary string) is determined and visualized using a color-coded plot.

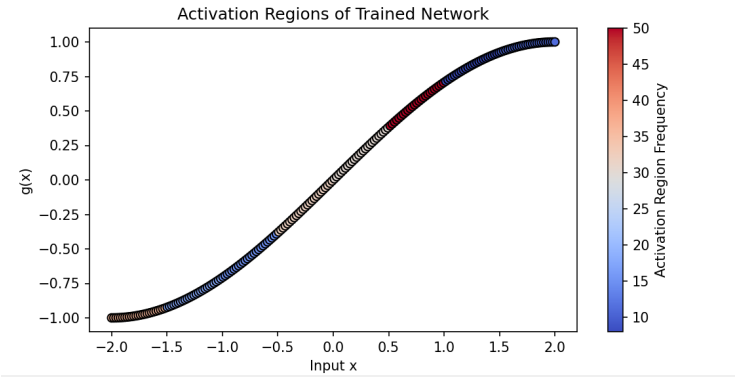

Figure 5: Activation Regions of Trained Network

(3) Quantifying activation pattern changes using Hamming distance

To measure network dynamics during training:

- Compute activation patterns at the start and end of each epoch.
- Compute the Hamming distance between the binary activation patterns of all training samples.
- Plot the total Hamming distance per epoch to observe pattern stability.
- Verify if the activation pattern stabilizes over time.

Problem 5: Gradient Norms and Classification Stability

(1) Training a Convolutional Neural Network on MNIST

A CNN is trained on a subset of MNIST to analyze gradient norms and classification error over time.

Network Design:

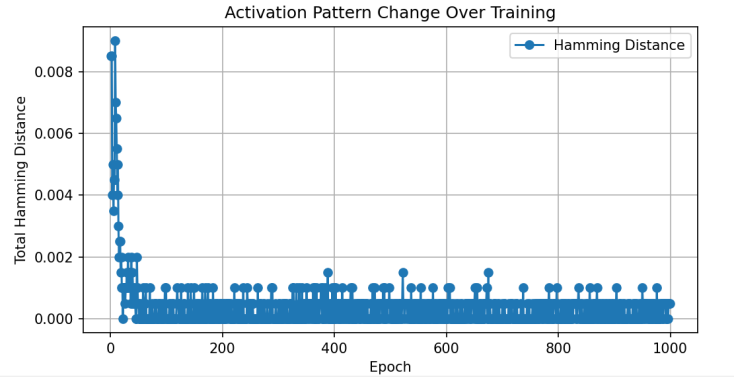

Figure 6: Activation Pattern Change Over Training

- Convolutional layer with ReLU activation.
- Fully connected output layer for classification.
- Cross-Entropy loss function.
- Adam optimizer with learning rate 0.001.
- Training over 25 epochs.

(2) Gradient Norms and Classification Error Over Time

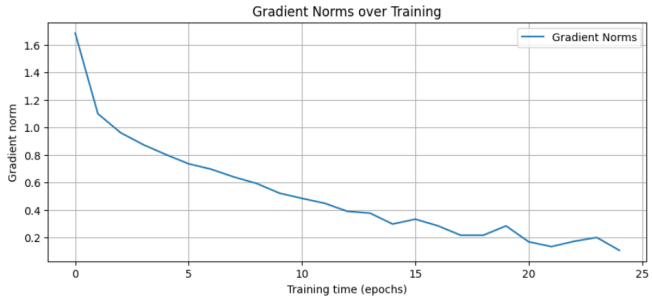

Figure 7: Gradient Norms Over Training

(3) Explanation for Stability Despite Large Gradients

- Despite large gradient norms, classification error stabilizes because useful representations have been learned early.

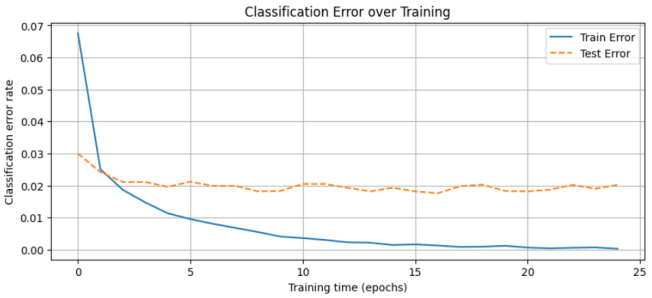

Figure 8: Classification Error Over Training

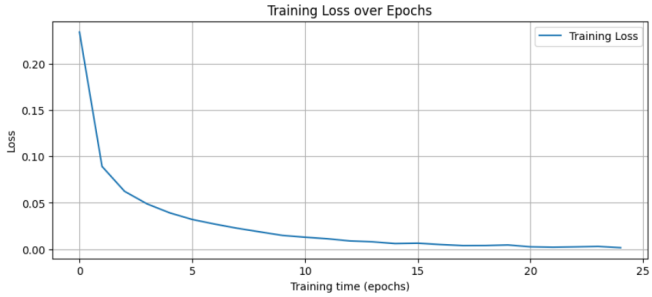

Figure 9: Training Loss Over Epochs

- Adaptive optimization methods like Adam prevent large parameter updates, reducing drastic model changes.
- The model fine-tunes decision boundaries with minimal parameter shifts in later epochs.

(4) Role of the Second-Order Term

- Second-order terms (Hessian-based approximations) help estimate curvature for optimization.
- They stabilize training by adjusting step sizes dynamically, ensuring convergence.
- This prevents excessive oscillations when gradient norms are large, improving generalization.