Understanding Neural Networks with Matrix Algebra

Dr. Gerardo E. Soto

Jornadas de Estadistica 2024 - Valdivia

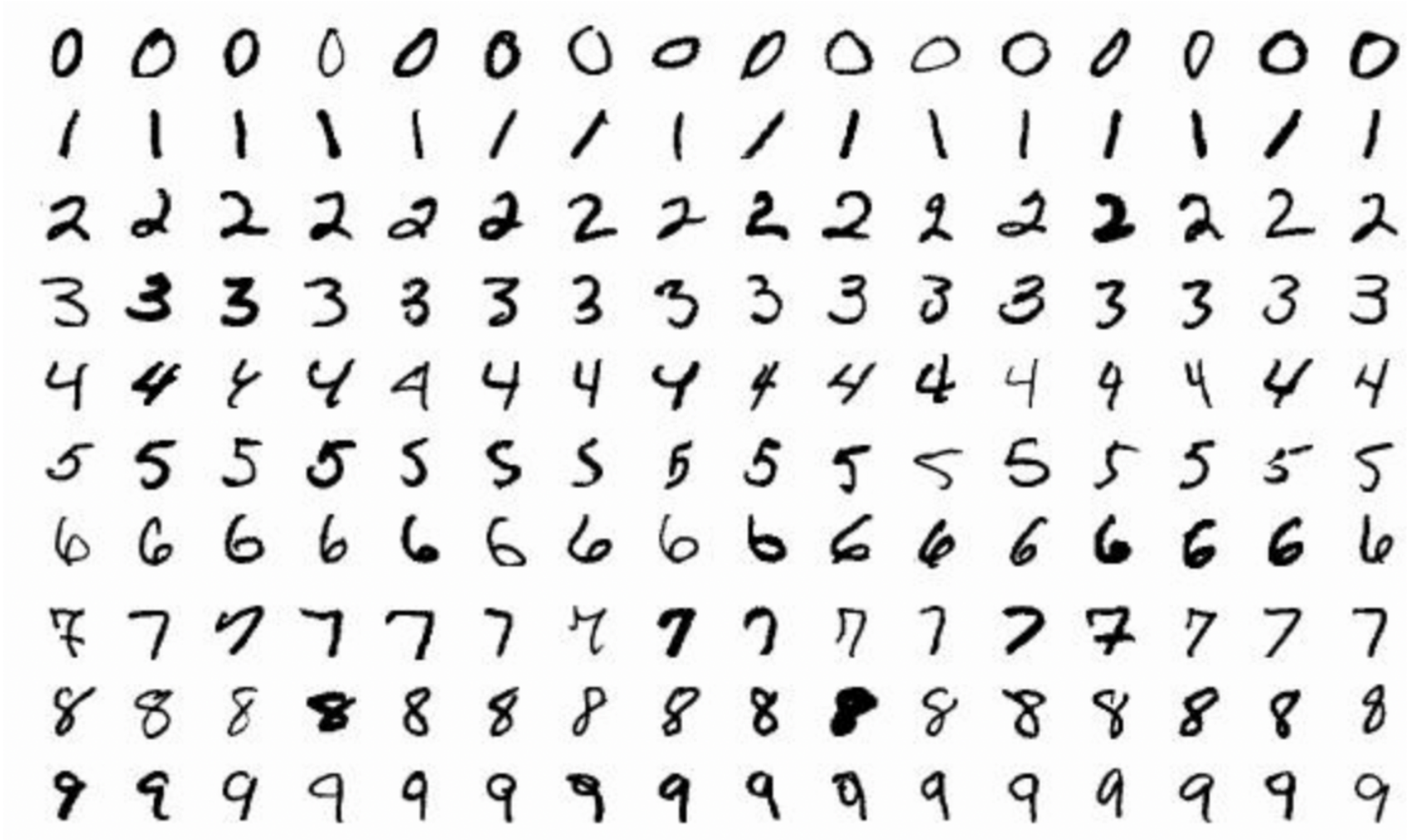
LAAR
Universidad Austral de Chile
Conocimiento y Naturaleza
Lab of Assisted Artificial Recognition

Universidad Austral de Chile
Conocimiento y Naturaleza
Instituto de Estadística

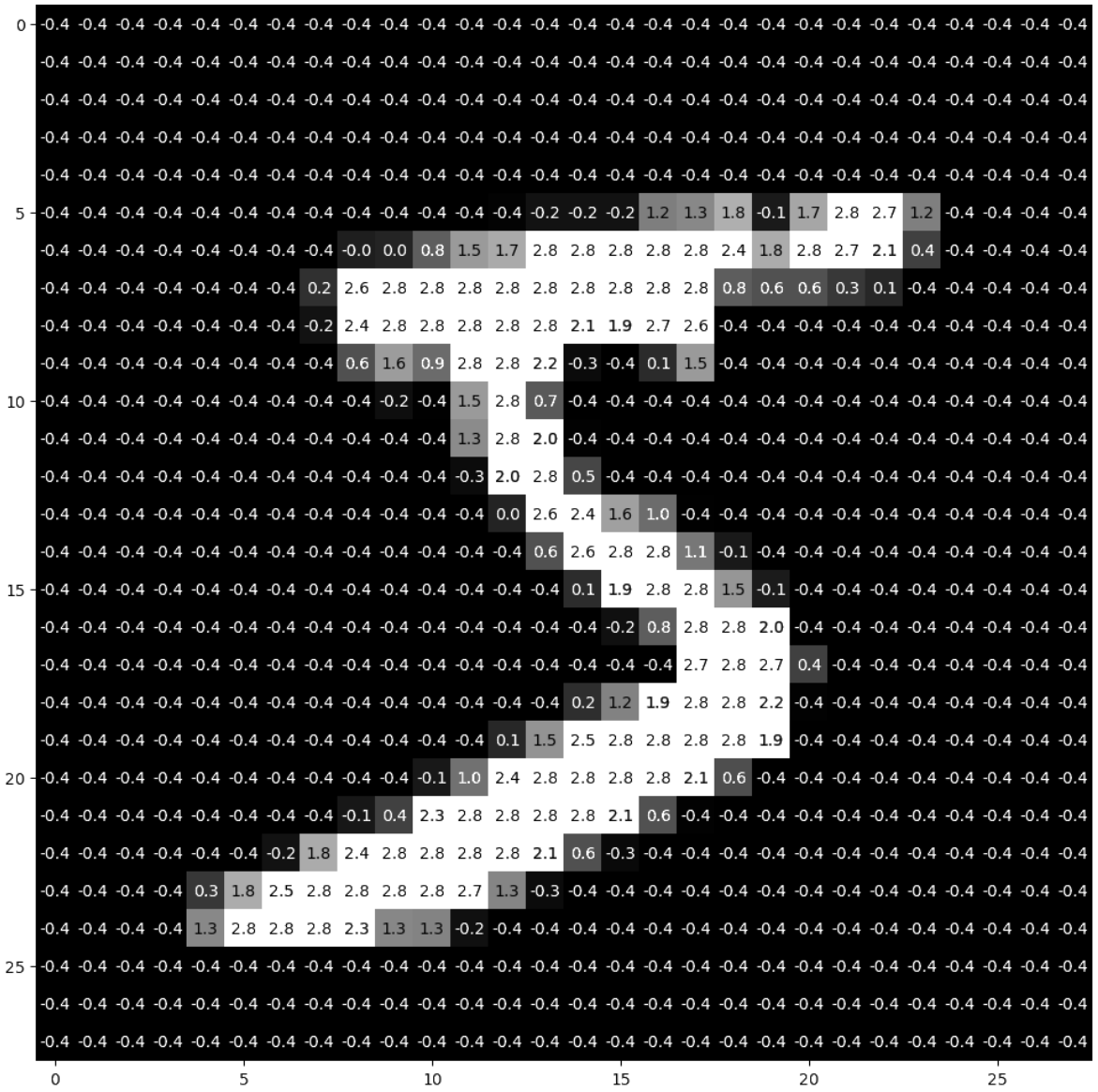
Introduction to Neural Networks

- Neural networks are computational models inspired by the human brain.
- They consist of interconnected layers of nodes (neurons).
- Commonly used for tasks like image recognition, natural language processing, and more.

Input Representation: MNIST Dataset



Input Representation: MNIST Dataset



Input Representation: MNIST Dataset

- The MNIST dataset contains 28x28 pixel images of handwritten digits.
- Each image is flattened into a vector of size 784 (28x28).
- Example of an input vector:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{784} \end{bmatrix}$$

Dense Network Architecture

- A dense (fully connected) network consists of layers where each neuron is connected to every neuron in the previous layer.
- Layers: Input Layer, Hidden Layers, Output Layer.

$$y = f(Wx + b)$$

where:

- W is the weight matrix,
- b is the bias vector,
- f is the activation function.

Weights in Neural Networks

- Weights determine the strength of the connection between neurons.
- Initialized randomly and updated during training.

$$W = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \dots & \dots & \dots & \dots \\ w_{m1} & w_{m2} & \dots & w_{mn} \end{bmatrix}$$

Activation Functions

Activation functions introduce non-linearity into the model. Common types include:

- Sigmoid:

$$f(x) = \frac{1}{1 + e^{-x}}$$

- ReLU (Rectified Linear Unit):

$$f(x) = max(0, x)$$

- Softmax (for multi-class classification):

$$f(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Forward Pass

Given an input vector x:

$$h = Wx + b$$

Apply activation function:

$$a = f(h)$$

Output layer calculation:

$$y = W_{\text{out}} a + b_{\text{out}}$$

Final output after activation:

$$y_{\text{final}} = f(y)$$

Backpropagation

Backpropagation is used to minimize the loss function (a.k.a. cost or error) by updating weights (It comes from "loss" of performance or accuracy). It involves:

- Calculating the loss (cross-entropy loss).

$$L(y, y_{\text{pred}}) = -\sum(y_i log(y_{\text{pred},i}))$$

y: true label or target value. It is often represented as a one-hot encoded vector for classification tasks.
y_{pred}: predicted probability distribution output by the model for each class.
y_{i}: ith element of the true label vector.
$y_{\text{pred,i}}$: predicted probability for the ith class.

Backpropagation

- Computing gradients with respect to weights:

$$\frac{\partial L}{\partial W}$$

This provides a direction to adjust the weights to reduce the loss. Note that this is a vector of gradients, one for each weight.

A positive gradient indicates that increasing the weight will increase the loss, while a negative gradient indicates that decreasing it will reduce the loss.

Backpropagation

- Updating weights using gradient descent:

$$W := W - \eta \frac{\partial L}{\partial W}$$

where η is the learning rate.

The learning rate is a hyperparameter that determines how large of a step we take in the direction of the negative gradient.

This is done iteratively, applying this update rule across multiple iterations (epochs), aiming to converge towards a set of weights that minimizes our the function.

Backpropagation procedure

The backpropagation algorithm consists of two main steps:

1. Forward Pass:
   - Compute activations for all layers.
2. Backward Pass:
   - Compute gradients from output layer to input layer using the chain rule.

$$\delta^L = (y_{\text{pred}} - y) * f'(z^L)$$

δ^L: Error term for the output layer.
y_{pred}: Predicted output from forward pass.
y: Actual target values.
$f'(z^L)$: Derivative of the activation function at layer L.

$$\delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} * f'(z^{(l)})$$

Backpropagation in Detail

Compute gradients for weights:

$$W^{(l)} := W^{(l)} - \eta * \nabla W^{(l)}$$

where $l = 1, 2, ..., L - 1$.

$\nabla W^{(l)} = a^{(l-1)}(\delta^{(l)})^T$: Gradient of weights with respect to loss.

$a^{(l-1)}$ represents the activations (or outputs) from the previous layer.

Summary and Conclusion

Neural networks leverage matrix algebra for efficient computation.

Key components include:

1. - Input Representation: Flattened images as vectors.
2. - Weights: Adjusted during training to minimize loss.
3. - Activation Functions: Introduce non-linearity.
4. - Backpropagation: Efficiently updates weights based on error.

This process repeats for multiple epochs over the training dataset, continuously refining weights to minimize loss.

Forward Pass Example with MNIST Digit '3'

Given an input vector $x = [0, 0, ..., 1, ..., 0]^T$:

- Assume weights for a single layer:

$$W = \begin{bmatrix} 0.2 & -0.5 \\ 0.3 & 0.8 \end{bmatrix}, b = \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix}$$

- Compute hidden layer output:

$$h = Wx + b$$

$$h = \begin{bmatrix} 0.2 & -0.5 \\ 0.3 & 0.8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix}$$

- Apply activation function:

$$a = f(h)$$

Calculating Loss for a given vector

Assume our target output for digit '3' is:

$$y_{\text{true}} = [0, 1, 0, ..., 0]$$

- Predicted output after softmax:

$$y_{\text{pred}} = [p_1, p_2, p_3, ..., p_n]$$

- Calculate loss using cross-entropy:

$$L(y_{\text{true}}, y_{\text{pred}}) = -\sum(y_{\text{true},i} log(y_{\text{pred},i}))$$

For example:

$$L = -[0 * log(p_1) + 1 * log(p_2) + ... + 0 * log(p_n)]$$

This simplifies to:

$$L = -log(p_2)$$

Backpropagation: Weight Updates Example

Using gradient descent to update weights based on loss L:

- Compute gradients:

$$\nabla L = [\partial L/\partial W_{11}, \partial L/\partial W_{12}, ...]$$

Assuming simple gradients for illustration:

$$\nabla W = \begin{bmatrix} -0.01 & 0.02 \\ 0.03 & -0.01 \end{bmatrix}$$

- Update weights with learning rate $\eta = 0.1$:

$$W := W - \eta * \nabla W$$

$$W_{\text{new}} = W_{\text{old}} + \begin{bmatrix} 0.002 & -0.002 \\ -0.003 & 0.001 \end{bmatrix}$$

$$W_{\text{new}} = \begin{bmatrix} 0.198 & -0.498 \\ 0.303 & 0.801 \end{bmatrix}$$

References:

- Goodfellow, I., Bengio, Y., Courville, A. (2016). Deep Learning. MIT Press.
- Bishop, C. M. (2006). Pattern Recognition and Machine Learning. Springer.

Open the link and press the Play button!