Lecture 2: Deep Learning - Introduction and Applications

Isaiah Hull[1,2]

[1]BI Norwegian Business School
[2]CogniFrame

October 31, 2023

Introduction

Lecture 2: Overview

1. Introduction to Deep Learning.
2. Deep Learning Applications.

1. Introduction to Deep Learning

Introduction to Deep Learning

Overview

- Based on Ng and Ma (2023).
  - https://cs229.stanford.edu/main_notes.pdf
- Overview of neural networks.
- Vectorization.
- Backpropagation.

Supervised Learning and Model Types

- Supervised learning: predicting y from input x.
- Model: $h_\theta(x)$.
- Familiar examples:
  - Linear regression: $h_\theta(x) = \theta^\top x$.
  - With feature map: $h_\theta(x) = \theta^\top \phi(x)$.

Supervised Learning and Model Types

- Common feature is linearity in parameters θ.
- Eventually consider non-linear models in θ and x.
- Will focus on neural networks specifically.
- Initially, consider $h_\theta(x)$ as an abstract non-linear model.

Training and Regression with Non-Linear Models

- Terminology differs from econometrics.
- Training examples (observations): $\left\{\left(x^{(i)}, y^{(i)}\right)\right\}_{i=1}^{n}$.
- Define a nonlinear model and an associated loss or cost function.
- Regression problem: continuous target (dependent variable).

Training and Regression with Non-Linear Models

- The output is a real number ($y^{(i)} \in \mathbb{R}$).
- Model also outputs a real number: $h_\theta(x) \in \mathbb{R}$.
- Loss function example: least squares.

Regression and Mean-Square Cost Function

- i-th example (observation) cost:

$$J^{(i)}(\theta) = \frac{1}{2}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

- Mean-square cost for entire dataset (sample):

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} J^{(i)}(\theta)$$

Regression and Mean-Square Cost Function

- Similar to OLS but:
  - Average, rather than sum of squared residuals.
  - Doesn’t change local/global minima.
  - Different parameterization for $h_\theta(x)$.
- “Loss" and “cost" used interchangeably.

Binary Classification and Loss Function

- Inputs: $x \in \mathbb{R}^d$.
- Parameterized model: $\bar{h}_\theta : \mathbb{R}^d \to \mathbb{R}$.
- Output (logit): $\bar{h}_\theta(x) \in \mathbb{R}$.
- Logistic function $g(\cdot)$ to convert logit:

$$h_\theta(x) = g\left(\bar{h}_\theta(x)\right) = \frac{1}{1 + \exp\left(-\bar{h}_\theta(x)\right)}$$

Binary Classification and Loss Function

- Conditional distribution of y:

$$P(y = 1 \mid x; \theta) = h_\theta(x)$$
$$P(y = 0 \mid x; \theta) = 1 - h_\theta(x)$$

- Negative likelihood loss function:

$$J^{(i)}(\theta) = -\log p\left(y^{(i)} \mid x^{(i)}; \theta\right) = \ell_{\text{logistic}}\left(\bar{h}_\theta\left(x^{(i)}\right), y^{(i)}\right)$$

Loss Functions

- Total loss:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} J^{(i)}(\theta)$$

- Multi-class classification:
  - Response variable y can be k values: $y \in \{1, 2, \ldots, k\}$.
  - Parameterized model: $\bar{h}_\theta : \mathbb{R}^d \to \mathbb{R}^k$.

Multi-Class Classification

- Outputs (logits): $\bar{h}_\theta(x) \in \mathbb{R}^k$.
- Use softmax for probability vector:

$$P(y = j \mid x; \theta) = \frac{\exp\left(\bar{h}_\theta(x)_j\right)}{\sum_{s=1}^{k} \exp\left(\bar{h}_\theta(x)_s\right)}$$

Multi-Class Classification

- i-th example loss (negative log-likelihood):

$$J^{(i)}(\theta) = \ell_{\text{ce}}\left(\bar{h}_\theta\left(x^{(i)}\right), y^{(i)}\right)$$

- Average of loss for training examples:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} J^{(i)}(\theta)$$

- Exponential family and non-linear parameterization.

Optimizers and SGD

- Common Optimizers: gradient descent (GD), stochastic gradient descent (SGD), and variants.
- Gradient descent update rule:

$$\theta := \theta - \alpha \nabla_{\theta} J(\theta)$$

- Learning Rate (α): controls step size.
- Modification of gradient descent: stochastic gradient descent.

Introduction to Deep Learning

Algorithm 1 Stochastic Gradient Descent

Hyperparameter: learning rate α, number of total iteration n_{iter}.
Initialize θ randomly.
for $i = 1$ to n_{iter} **do**
    Sample j uniformly from $\{1, \ldots, n\}$, and update θ by

$$\theta := \theta - \alpha \nabla_\theta J^{(j)}(\theta) \tag{7.9}$$

Source: Ng and Ma (2023).

Hardware Parallelization and Mini-batch SGD

- Computing gradient of B (batch size) examples simultaneously often faster due to hardware parallelization.
- Mini-batch version of SGD prevalent in deep learning.
- Variants of SGD and mini-batch SGD exist with different sampling schemes.

Introduction to Deep Learning

Algorithm 2 Mini-batch Stochastic Gradient Descent

Hyperparameters: learning rate α, batch size B, # iterations n_{iter}.
Initialize θ randomly
for $i = 1$ to n_{iter} **do**
    Sample B examples $j_1, \ldots, j_B$ (without replacement) uniformly from $\{1, \ldots, n\}$, and update θ by

$$\theta := \theta - \frac{\alpha}{B} \sum_{k=1}^{B} \nabla_\theta J^{(j_k)}(\theta) \tag{7.10}$$

Source: Ng and Ma (2023).

Steps in Training a Deep Learning Model

1. Define a neural network parametrization $h_\theta(x)$.

2. Implement the backpropagation algorithm for efficient gradient computation of $J^{(j)}(\theta)$.

3. Execute SGD, mini-batch SGD, or other gradient-based optimizers with $J(\theta)$.

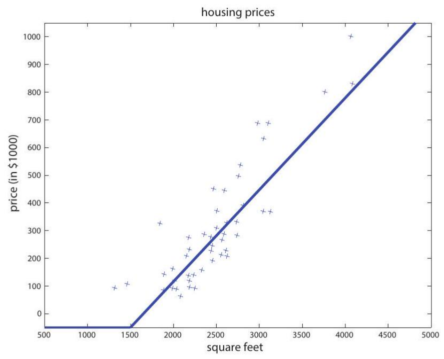
Neural Networks: Introduction

- Broad type of non-linear models: $\bar{h}_\theta(x)$.
- Combines matrix multiplications with element-wise non-linear operations.
- Regression: $h_\theta(x) = \bar{h}_\theta(x)$.
- Classification:
  - Binary: $h_\theta(x) = 1/\left(1 + \exp\left(-\bar{h}_\theta(x)\right)\right)$
  - Multi-class: $h_\theta(x) = \text{softmax}\left(\bar{h}_\theta(x)\right)$

Neural Network with a Single Neuron: The Housing Example

- Example in this unit: predict house price from its size.
- Starting point: fit a straight line.
- New requirement: disallow negative prices, introducing a “kink."
- How can we represent a function with a kink as $\bar{h}_\theta(x)$?

Introduction to Deep Learning



Source: Ng and Ma (2023).

Parameterization and Activation Functions

- Define $h_\theta(x)$:

$$\bar{h}_\theta(x) = \max(wx + b, 0),$$

$$\text{where } \theta = (w, b) \in \mathbb{R}^2$$

Parameterization and Activation Functions

- Rectified linear unit (ReLU):

$$\mathsf{ReLU}(t) \equiv \max\{t, 0\}$$

- ReLU is an example of an activation function.
- Single neuron due to single non-linear activation function.

Basic Terminology

- b: often referred to as the “bias."
- w: known as the weight vector.
- Can extend to multi-neuron case.

Neural Network with a Single Neuron for Multidimensional Input

$$\bar{h}_\theta(x) = \mathrm{ReLU}\left(w^\top x + b\right) \tag{1}$$

- Where: $w \in \mathbb{R}^d$, $b \in \mathbb{R}$, and $\theta = (w, b)$.

Stacking Neurons

- Neural networks can be made more complex by stacking neurons.
- Each neuron passes its output as input to the next neuron.
- Allows for flexibility in modeling complex functions.

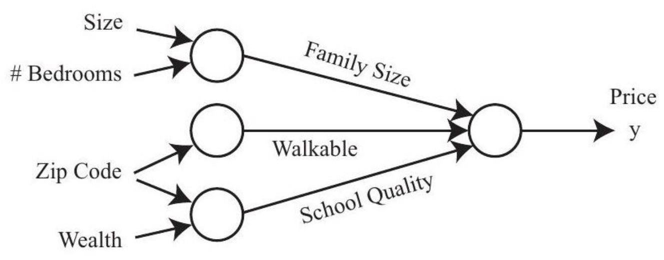
Deepening the Housing Example

- New features: size, number of bedrooms, zip code, wealth of neighborhood.
- Neural network have modular structure, which allows us to stack them to create complex structures.
- Individual neurons are fundamental building block.

Deriving Features for Housing Price Prediction

- Family size: function of size of the house and number of bedrooms.
- Walkability: inferred from zip code.
- School quality: based on zip code and neighborhood wealth.
- House price: depends on family size, walkability, and school quality.

Small Neural Network for Predicting Housing Prices



Source: Ng and Ma (2023).

Input and Hidden Units

- Input features: x_1, x_2, x_3, x_4.
- Intermediate variables (hidden units): a_1 (family size), a_2 (walkable), a_3 (school quality)

$$
\begin{aligned}
a_1 &= \mathrm{ReLU}\left(\theta_1 x_1 + \theta_2 x_2 + \theta_3\right) \\
a_2 &= \mathrm{ReLU}\left(\theta_4 x_3 + \theta_5\right) \\
a_3 &= \mathrm{ReLU}\left(\theta_6 x_3 + \theta_7 x_4 + \theta_8\right)
\end{aligned}
$$

- Parameters: $(\theta_1, \cdots, \theta_8)$.

Output Parameterization

- Output representation: $\bar{h}_\theta(x)$.

$$\bar{h}_\theta(x) = \theta_9 a_1 + \theta_{10} a_2 + \theta_{11} a_3 + \theta_{12}$$

- Parameters: $\theta = (\theta_1, \cdots, \theta_{12})$.
- Complex function of x with parameters θ.
- Try to learn θ.

Inspiration from Biological Neural Networks

- Artificial neural networks inspired by biological ones.
- Hidden units: analogous to biological neurons.
- Parameters θ_i: correspond to synapses.
- Uncertain similarity: deep artificial networks vs. biological ones.
- Open question: human brain learning mechanism vs. backpropagation in artificial networks.

Two-layer Fully-Connected Neural Networks

- Intermediate variables are functions of all $x_1, \ldots, x_4$.
- This setup avoids the need for prior assumptions:

$$a_1 = \mathrm{ReLU}\left(w_1^\top x + b_1\right)$$
$$a_2 = \mathrm{ReLU}\left(w_2^\top x + b_2\right)$$
$$a_3 = \mathrm{ReLU}\left(w_3^\top x + b_3\right)$$

- The network is fully-connected: all a_i's depend on all x_i's.

General Two-layer Fully-Connected Neural Network

- For input $x \in \mathbb{R}^d$ and m hidden units.
- Connection to previous slide: we generalize to d dimensions.

$$
\begin{aligned}
\forall j \in [1, \ldots, m], \quad z_j &= w_j^{[1]^\top} x + b_j^{[1]} \\
a_j &= \mathrm{ReLU}\left(z_j\right), \\
a &= [a_1, \ldots, a_m]^\top \\
\bar{h}_\theta(x) &= w^{[2]^\top} a + b^{[2]}
\end{aligned}
$$

- Notation: a is a column vector, and indices $^{[1]}$ and $^{[2]}$ distinguish parameters.

Vectorization in Neural Networks

- Simplify neural network expressions using matrix and vector notations.
- Speed up implementation by avoiding for loops.
- Utilize matrix algebra and optimized numerical packages.
- Modern networks need vectorization for efficiency.

Vectorizing the Neural Network: Weight Matrix

- Define weight matrix $W^{[1]}$ by concatenating all vectors $w_j^{[1]}$.
- Dimension of $W^{[1]}$: $\mathbb{R}^{m \times d}$:

$$W^{[1]} = \begin{bmatrix} -{w_1^{[1]}}^\top - \\ -{w_2^{[1]}}^\top - \\ \vdots \\ -{w_m^{[1]}}^\top - \end{bmatrix}$$

Computing z in Vectorized Form

- Use matrix-vector multiplication.
- Define z: $[z_1, \ldots, z_m]^\top \in \mathbb{R}^m$.

$$z = W^{[1]}x + b^{[1]}$$

Vectors as Column Vectors

- By default, vectors in $\mathbb{R}^d$ are viewed as column vectors.
- Vectors can also be treated as $d \times 1$ matrices.

Computing Activations

- Activations are computed using the ReLU function.
- Element-wise non-linear application.
- Element-wise ReLU is parallelized efficiently.

$$a = \mathrm{ReLU}(z)$$

Summarizing the Model

- Define $W^{[2]}$: $\left[w^{[2]^\top} \right] \in \mathbb{R}^{1 \times m}$.
- Use weight matrices for efficient computation.

$$a = \mathrm{ReLU}\left(W^{[1]} x + b^{[1]} \right)$$
$$\bar{h}_\theta(x) = W^{[2]} a + b^{[2]}$$

Structure of a Neural Network

- $W^{[1]}$ and $W^{[2]}$: weight matrices for first and second layers.
- $b^{[1]}$ and $b^{[2]}$: biases for the first and second layers.
- Activation a is the hidden layer.
- A two-layer neural network is a one-hidden-layer neural network.

Multi-layer Neural Networks

- Generalizes concept to multiple layers.
- Activation: $a^{[k]} = \mathrm{ReLU}(W^{[k]} a^{[k-1]} + b^{[k]})$.
- Output: $\bar{h}_\theta(x) = W^{[r]} a^{[r-1]} + b^{[r]}$.

Activation Functions

- Functions applied element-wise introducing non-linearity.
- Example functions:
  - Sigmoid
  - Tanh
  - Leaky ReLU
  - GELU
  - Softplus

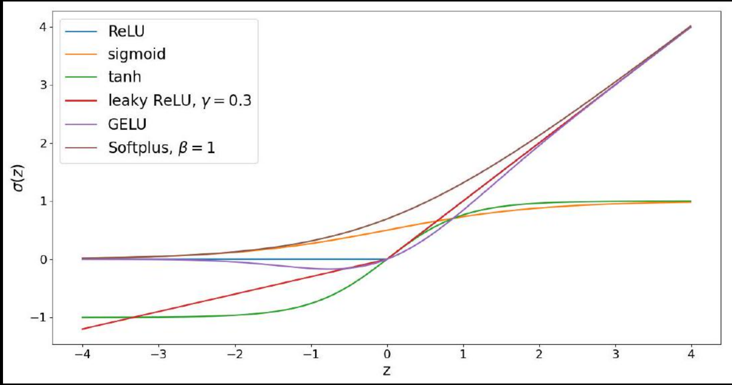
Linearity vs Non-Linearity

- Why not $\sigma(z) = z$?
- With linear activation, multiple layers collapse to a single matrix product:

$$\tilde{W}x \text{ where } \tilde{W} = W^{[2]}W^{[1]}.$$

- Importance of non-linear activation functions: represent complex relationships.

Activation Functions



Source: Ng and Ma (2023).

Feature Maps in Traditional Machine Learning

- Represent non-linear functions: $\theta^\top \phi(x)$.
  - θ: parameters; $\phi(x)$: feature map.
- Performance depends on the feature map choice.
- Process of choosing: feature engineering.

Deep Learning as Feature Learning

- Feature map is learned, not handcrafted.
- Let β be parameters (excluding final layer).
- Abstract: $a^{[r-1]} = \phi_\beta(x)$.
- Model: $\bar{h}_\theta(x) = W^{[r]}\phi_\beta(x) + b^{[r]}$.

Training and Optimization

- When β is fixed, $\phi_\beta(\cdot)$ is a feature map.
- During training, β, $W^{[r]}$, and $b^{[r]}$ are optimized.
- Learn the model and feature map simultaneously.
- Less dependence on domain knowledge and feature engineering.

Deep Learning Representations

- Second to last layer $a^{[r]}$ is the "learned feature."
  - E.g., in house pricing, neural network might learn "family size."
- Learned representations can be transferred across datasets.
  - Might be complex and difficult to interpret.

Interpretability Challenge

- Neural networks can be viewed as a “black box."
- Difficult to discern the features they discover.
- This poses challenges for interpretability.

Modules in Modern Neural Networks

- Modern neural networks are more sophisticated.
- Networks incorporate layers and building blocks.
- Will examine building blocks.

Matrix Multiplication as a Building Block

- Matrix multiplication serves as a primary building block:

$$\mathrm{MM}_{W,b}(z) = Wz + b$$

- W denotes the weight matrix.
- b is the bias vector.
- z is the input.

MLP: Composition of Modules

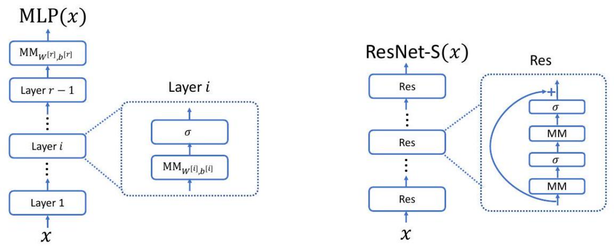
- MLP is a sequence of matrix multiplication modules interwoven with nonlinear activation modules:

$$\mathsf{MLP}(x) = \mathsf{MM}(\sigma(\mathsf{MM}\,\sigma(\cdots \mathsf{MM}(x))))$$

- Each module typically has its unique set of parameters.

Larger Modules from Smaller Ones

- A single 'layer' often encompasses an activation function, σ, and a matrix multiplication module, MM.



Source: Ng and Ma (2023).

Residual Connections

- ResNets have revolutionized vision applications
- A residual block is given as follows:

$$\mathsf{Res}(z) = z + \sigma(\mathsf{MM}(\sigma(\mathsf{MM}(z))))$$

- A simplified ResNet:

$$\mathsf{ResNet\text{-}S}(x) = \mathsf{MM}(\mathsf{Res}(\mathsf{Res}(\cdots \mathsf{Res}(x))))$$

Layer Normalization

- Layer normalization, LN, is pivotal post-nonlinear activations. It transforms:

$$\mathsf{LN-S}(z) = \begin{bmatrix} \frac{z_1 - \hat{\mu}}{\hat{\sigma}} \\ \vdots \\ \frac{z_m - \hat{\mu}}{\hat{\sigma}} \end{bmatrix}$$

- However, merely having zero mean and standard deviation 1 might not be ideal. Hence:

$$\mathsf{LN}(z) = \beta + \gamma \cdot \mathsf{LN-S}(z)$$

Computational Graph Representation

- Each computation visualized as a computational graph.
- Nodes represent operations, variables, or functions.
- Example: $y = \sigma(Wx + b)$
  - x: input
  - W: weight matrix
  - b: bias
  - σ: activation function
- Equations can be broken down into smaller operations.

Layer Normalization

- Relies on chain rule.
- If $f(g(h(x)))$, then:

$$\frac{df}{dx} = \frac{df}{dg} \times \frac{dg}{dh} \times \frac{dh}{dx}$$

- Compute local gradient and multiply with gradient from next operation.

Forward and Backward Passes

1. Forward Pass:
   - Compute output layer by layer.
   - Standard computation pass.
2. Backward Pass:
   - Compute gradient of loss with respect to each neuron's output.
   - Continue layer-by-layer in reverse.

Weight Update

- Update weights using optimization algorithm.
- Adjust weights to decrease error:

$$W_{new} = W_{old} - \alpha \times \frac{\partial L}{\partial W}$$

- where $L =$ loss and α = learning rate.

Challenges with Backpropagation

- Vanishing gradient: gradients diminish to near zero.
- Exploding gradient: gradients become too large.
- Local minima: possible to get stuck in suboptimal solutions.

Improvements and Variants

- Stochastic and mini-batch GD: update after a small set.
- Momentum: use previous weight change for current update.
- Advanced Optimizers: Adam, RMSprop.
- Regularization techniques: dropout, L1/L2 regularization, early stopping.

Overview

- Review of the basic chain rule.
- Introduction to the general strategy for backpropagation.
- Computation of the backward function for basic modules in neural networks.
- Concrete backprop algorithm for MLPs.

Partial Derivatives in NNs

- Scalar variable J depending on variables z.
- Notation: $\frac{\partial J}{\partial z}$.
- Dimension of $\frac{\partial J}{\partial z}$ is same as z.
- Example: If $z \in \mathbb{R}^{m \times n}$, then $\frac{\partial J}{\partial z} \in \mathbb{R}^{m \times n}$.

Remark

- When both J and z are not scalars, partial derivatives can become a matrix or tensor.
- Computationally expensive and rarely explicitly constructed.
- Focus is on derivatives of scalar function w.r.t vectors, matrices, or tensors.

Chain Rule

- Reviewing the chain rule:

$$
\begin{aligned}
z &\in \mathbb{R}^m \\
u &= g(z) \in \mathbb{R}^n \\
J &= f(u) \in \mathbb{R}.
\end{aligned}
$$

- Final variable J is a scalar.

Chain Rule (cont.)

$$u = (u_1, \ldots, u_n)$$

$$g(z) = (g_1(z), \cdots, g_n(z))$$

- Chain rule provides:

$$\forall i \in \{1, \ldots, m\}, \quad \frac{\partial J}{\partial z_i} = \sum_{j=1}^{n} \frac{\partial J}{\partial u_j} \cdot \frac{\partial g_j}{\partial z_i}$$

Vectorized Notation

- When z and u are vectors:

$$\frac{\partial J}{\partial z} = \begin{bmatrix} \frac{\partial g_1}{\partial z_1} & \cdots & \frac{\partial g_n}{\partial z_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial g_1}{\partial z_m} & \cdots & \frac{\partial g_n}{\partial z_m} \end{bmatrix} \cdot \frac{\partial J}{\partial u}$$

Backward Function and Linear Map

- The backward function is always a linear map from $\frac{\partial J}{\partial u}$ to $\frac{\partial J}{\partial z}$.
- The mapping can depend on z in complex ways.
- The matrix on the RHSis the transpose of the Jacobian matrix of g.

Advantages

- Avoids in-depth discussion about Jacobian matrices.
- Convenient in cases like $z \in \mathbb{R}^{r \times s}$, giving:

$$\forall i, k, \quad \frac{\partial J}{\partial z_{ik}} = \sum_{j=1}^{n} \frac{\partial J}{\partial u_j} \cdot \frac{\partial g_j}{\partial z_{ik}}.$$

Key Interpretation of the Chain Rule

- Chain rule provides a way to compute $\frac{\partial J}{\partial z}$ from $\frac{\partial J}{\partial u}$.
- Only requires knowledge about g.

Backward Function Notation

- Denoted as $\mathcal{B}[g, z]$.
- For fixed z, $\mathcal{B}[g, z]$ is a linear map from $\mathbb{R}^n$ to $\mathbb{R}^m$.

Backward Function as Matrix

- Viewed as a matrix, but depends on changing z.
- Inputs: z (input to g) and v (gradient w.r.t to output of g).
- Outputs: gradient of J w.r.t z.

General Strategy of Backpropagation

- Neural networks: complex compositions of building blocks.
- Modules: MM, σ, Conv2D, LN, and others.
- Losses are also viewed as modules.

Example of Modular Composition

- Example: $J = M_k\left(M_{k-1}\left(\cdots M_1(x)\right)\right)$.
- Modules involve parameters or fixed operations.
- Each M_i involves a set of parameters $\theta^{[i]}$.

Intermediate Variables

- Introduce intermediate variables.

$$
\begin{aligned}
u^{[0]} &= x \\
u^{[1]} &= M_1\left(u^{[0]}\right) \\
u^{[2]} &= M_2\left(u^{[1]}\right) \\
&\vdots \\
J = u^{[k]} &= M_k\left(u^{[k-1]}\right).
\end{aligned}
$$

Overview of Backpropagation

- Backpropagation consists of two passes:
  - Forward pass.
  - Backward pass.

Intermediate Variables

- Compute $u^{[1]}, \ldots, u^{[k]}$ from $i = 1, \ldots, k$, sequentially using the definition in (F).
- Save all intermediate variables $u^{[i]}$'s in memory.

Backward Pass

- Compute derivatives w.r.t intermediate variables: $\frac{\partial J}{\partial u^{[k]}}, \ldots, \frac{\partial J}{\partial u^{[1]}}$.
- Compute derivatives of parameters: $\frac{\partial J}{\partial \theta^{[i]}}$.
- Interleaved computations are possible.

Chain Rule in Backpropagation

Efficient computation of $\frac{\partial J}{\partial u^{[i-1]}}$ from $\frac{\partial J}{\partial u^{[i]}}$ and $u^{[i-1]}$:

$$\frac{\partial J}{\partial u^{[i]}} \Longrightarrow \frac{\partial J}{\partial u^{[i-1]}}.$$

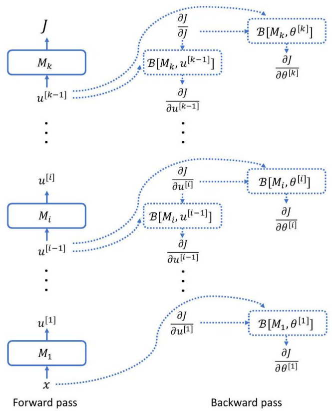
$$\frac{\partial J}{\partial u^{[i-1]}} = \mathcal{B}\left[M_i, u^{[i-1]}\right]\left(\frac{\partial J}{\partial u^{[i]}}\right).$$

Further Chain Rule Application

- Instantiating with $z = \theta^{[i]}$ and $u = u^{[i]}$:

$$\frac{\partial J}{\partial \theta^{[i]}} = \mathcal{B}\left[M_i, \theta^{[i]}\right]\left(\frac{\partial J}{\partial u^{[i]}}\right).$$

Further Chain Rule Application



Source: Ng and Ma (2023)

Computational Efficiency and Granularity

- Small modules have efficient backward functions.
- Backward functions of atomic modules like addition, multiplication, ReLU are computationally efficient.
- Neural networks as compositions of atomic operations.
- Modularize using matrix multiplication, layer norm, etc. for practicality.

Backward Functions for Basic Modules

- Compute the backward function for:
  - Basic module MM.
  - Activations σ.
  - Loss functions.

Introduction to Deep Learning

$$\mathrm{MM}_{W,b}(z) = Wz + b \tag{2}$$

$$\mathcal{B}[\mathrm{MM}, z](v) = W^\top v \in \mathbb{R}^m \tag{3}$$

$$\mathcal{B}[\mathrm{MM}, W](v) = vz^\top \in \mathbb{R}^{n \times m} \tag{4}$$

$$\mathcal{B}[\mathrm{MM}, b](v) = v \tag{5}$$

Computational Efficiency

- Computational efficiency for computing the backward function is $O(mn)$.
- Equivalent to evaluating the result of matrix multiplication up to a constant factor.

Backward Function for Activations

- Let $M(z) = \sigma(z)$ where σ is an element-wise activation function and $z \in \mathbb{R}^m$.
- We have:

$$\mathcal{B}[\sigma, z](v) = \sigma'(z) \odot v \in \mathbb{R}^m.$$

Notes on Activation Backward Function

- $\frac{\partial \sigma(z_j)}{\partial z_i} = 0$ when $j \neq i$.
- The backward function looks like $O(m^2)$ but is $O(m)$ in practice.
- Using smaller modules simplifies the process.

Backward Function for Loss Functions

- Given a module M taking vector z and outputting a scalar, we have:

$$\mathcal{B}[M, z](v) = \frac{\partial M}{\partial z} \cdot v$$

- Examples:
  - Squared Loss: $\mathcal{B}\left[\ell_{\mathrm{MSE}}, z\right](v) = (z - y) \cdot v$
  - Logistics Loss: $\mathcal{B}\left[\ell_{\mathsf{logistic}}, t\right](v) = (1/(1 + \exp(-t)) - y) \cdot v$
  - Cross-Entropy Loss: $\mathcal{B}\left[\ell_{\mathrm{ce}}, t\right](v) = (\phi - e_y) \cdot v$

Back-propagation for MLPs

- If we are given backward functions for modules to evaluate loss, can compute gradient of loss with respect to hidden activations and parameters.

Back-propagation for MLPs: Forward Pass

- An r-layer MLP with logistic loss:

$$
\begin{aligned}
z^{[1]} &= \mathrm{MM}_{W^{[1]},b^{[1]}}(x), \\
&\vdots \\
J &= \ell_{\text{logistic}}\left(z^{[r]}, y\right).
\end{aligned}
$$

Back-propagation for MLPs: Backward Pass

- Compute the gradient of loss J w.r.t $z^{[r]}$:

$$\frac{\partial J}{\partial z^{[r]}} = \left(1/\left(1 + \exp\left(-z^{[r]}\right)\right) - y\right)$$

- Then, compute the gradient with respect to parameters $W^{[k+1]}$ and $b^{[k+1]}$ and others iteratively.

Algorithm 3: Back-propagation for Multi-layer Neural Networks

1. Forward pass. Compute and store the values of $a^{[k]}$'s, $z^{[k]}$'s, and J.
2. Backward pass. Compute the gradient of loss J with respect to $z^{[r]}$.
3. Iterate from $k = r - 1$ to 0 for the rest.

Backpropagation Equations

$$
\begin{aligned}
\frac{\partial J}{\partial a^{[k]}} &= \mathcal{B}\left[\sigma, a^{[k]}\right]\left(\frac{\partial J}{\partial z^{[k+1]}}\right) \\
&= W^{[k+1]^\top} \frac{\partial J}{\partial z^{[k+1]}} \\
\frac{\partial J}{\partial z^{[k]}} &= \mathcal{B}\left[\sigma, z^{[k]}\right]\left(\frac{\partial J}{\partial a^{[k]}}\right) \\
&= \sigma'\left(z^{[k]}\right) \odot \frac{\partial J}{\partial a^{[k]}}
\end{aligned}
$$

Vectorization Over Training Examples

- Avoid explicit loops by stacking examples in matrices.
- Each column in a matrix represents a training example.
- Efficient computation on hardware optimized for matrix operations.

Column Vector Notation

- Use broadcasting to add bias $b^{[1]}$ to each column of $W^{[1]}X$.

$$Z^{[1]} = W^{[1]}X + \tilde{b}^{[1]}$$

- Not necessary to construct $\tilde{b}^{[1]}$ explicitly.

Data Representation Mismatch

- Mismatch between deep learning packages (row vectors) and notation in papers (column vectors).
- Adjustments:
  - Columns become row vectors and vice-versa.
  - Matrices are transposed.
  - Order of matrix multiplication is flipped.
- Possible reason: natural inclination to multiply a matrix to a vector from the left.

2. Deep Learning Applications

Introduction to TensorFlow

Colab Tutorial

- Introduction to Deep Learning

References I

Ng, Andrew and Tengyu Ma (2023) “CS229 Lecture Notes,” June.