Natural Language Processing (NLP) and Large Language Models (LLMs)
Lecture 3-1: Machine learning

Chendi Wang (王晨笛)
chendi.wang@xmu.edu.cn

WISE @ XMU

2025 年 3 月 11 日

Adapted in part from 《动手学深度学习》 and 《AI and machine learning》.

Recap

- We use functions like SoftMax and sigmoid (activation functions) in Word2Vec.
- This makes Word2Vec a (simple) neural network.
- Let's start with the simplest linear models.

Recall: linear regression

- Consider a dataset , potentially drawn independently from some underlying distribution $\mathbb{P}$, $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$. Here , $\mathbf{x}_i \in \mathbb{R}^d$ is called features (or covariates) and $y_i \in \mathbb{R}$ is called labels (or targets).
- In the context of linear regression, we assume the relationship:

$$y_i = \mathbf{w}_*^T \mathbf{x}_i + b_* + \epsilon_i,$$

where $\mathbf{w}_* \in \mathbb{R}^d$ and $b_* \in \mathbb{R}$ are the "true parameters," and ϵ_i is a noise term.

- The least-squares regression problem is formulated as:

$$(\widehat{\mathbf{w}}, \widehat{b}) = \operatorname{argmin}_{(\mathbf{w},b)} \left\{ \frac{1}{n} \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x}_i - b)^2 \right\}.$$

- For a new feature vector $\mathbf{x} \in \mathbb{R}^d$, the predicted label is:

$$\widehat{y}(\mathbf{x}) = \widehat{\mathbf{w}}^T \mathbf{x} + \widehat{b}.$$

- Without loss of generality, we assume no bias term b for simplicity. (Why? subsume it to $\mathbf{w}$ or center the labels y_i)

Recall: closed-form representaion of the OLS

- Denote $\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_n]^T \in \mathbb{R}^{n \times d}$ and $\mathbf{y} = [y_1, \cdots, y_n]^T \in \mathbb{R}^n$.
- Then we can rewrite

$$\left\{ \frac{1}{n} \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 \right\} = \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 .$$

- If we assume $\mathbf{X}^T\mathbf{X}$ is invertible, then it holds

$$\widehat{\mathbf{w}} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{X}^T\mathbf{y}.$$

- Why?

$$\partial_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 = 0 \Rightarrow ?$$

- Calculating the inverse of a matrix is time-consuming when d is large (remedy: using SGD).

Loss functions

- Recall $\ell_{(\mathbf{w},b)}(\mathbf{z}_i) := (y_i - \mathbf{w}^T\mathbf{x}_i - b)^2$ with $\mathbf{z}_i = (\mathbf{x}_i, y_i)$. Here, $\ell_{(\mathbf{w},b)}$ is the least squares loss function.
- A loss function ℓ_θ, parameterized by θ (e.g., $\theta = (\mathbf{w}, b)$ above), maps a data point $\mathbf{z} = (\mathbf{x}, y)$ and a model parameter θ to a real number, measuring how well predictions match the actual target.
- Why does least squares loss measure prediction quality? (it estimates the conditional mean)

Train and test

- Train:

$$\widehat{\theta} = \operatorname{argmin}_\theta \qquad \frac{1}{n}\sum_{i=1}^{n} \ell_\theta(\mathbf{z}_i).$$

- Generalization error:

$$\mathbb{E}_{\mathbf{z}\sim\mathbb{P}}\ell_{\widehat{\theta}}(\mathbf{z}),$$

for $\mathbf{z} = (\mathbf{x}, y) \sim \mathbb{P}$ where $\mathbb{P}$ is the distribution of the training data (i.e.,$\mathbf{z}_i \sim \mathbb{P}$).

- Test: for another set of data points $\{\mathbf{z}_i^{\text{test}}\}_{i=1}^m \sim \mathbb{P}$, the testing error is

$$\frac{1}{m}\sum_{i=1}^{m} \ell_{\widehat{\theta}}(\mathbf{z}_i^{\text{test}}).$$

Data splitting

- There should be no overlap between the training data set and the testing data set (double dipping).
- demonstration (linear_data_spliting.ipynb)

Why least squares loss?

- In linear least squares regression, assume the true parameter $(\mathbf{w}_*, b_*)$ satisfies $\mathbb{E}[y|\mathbf{x}] = \mathbf{w}_*^T\mathbf{x} + b_*$ for $(\mathbf{x}, y)$ drawn from some distribution $\mathbb{P}$.
- Then,

$$\mathbb{E}[y|\cdot] = \operatorname{argmin}_f \quad \mathbb{E}_{(\mathbf{x},y)\sim\mathbb{P}}(y - f(\mathbf{x}))^2,$$

  where the infimum is over all square-integrable functions.
- Least squares loss is minimized by the conditional mean.
- If a predictor $\widehat{y}$ leads to a smaller least squares loss, then it is closer to the conditional mean.

From linear to non-linear

- The linear model is based on the function class

$$\mathcal{F} = \{f_{\mathbf{w},b}(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b; \mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R}\}.$$

- A linear model may be too simple for complex tasks (e.g., understanding and generating human language).
- If we use a general function class $\mathcal{F}$, how do we parameterize a function $f \in \mathcal{F}$?
- Why parameterize? With parameters, we can update them—most languages handle parameter updates better than function updates.

Neural Networks (Biology)

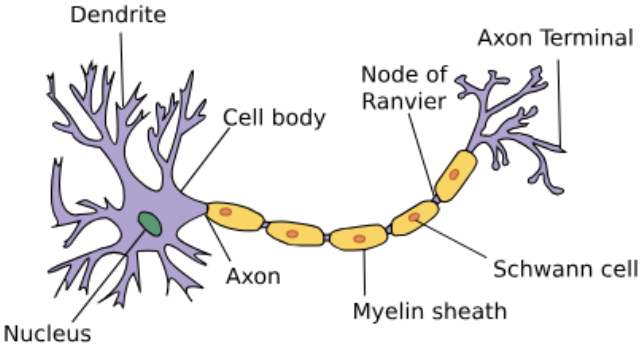


Figure: An example of human neural networks

Figure is from `https://d2l.ai/chapter_linear-regression/linear-regression.html`, the source is "Anatomy and Physiology" by the US National Cancer Institute's Surveillance, Epidemiology and End Results (SEER) Program

Artificial Neural Network (ANN)

- Each node is a **neuron**, inspired by biology.
- Neurons receive signals and pass them to the next layer.

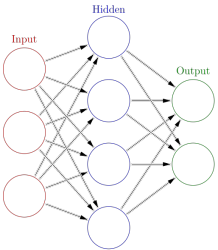


Figure from Wikipedia

Linear model as a neural network

- Input: $\mathbf{x} = [x_1, \ldots, x_d]$ with $x_i \in \mathbb{R}$.
- Each neuron i has a weight $w_i \in \mathbb{R}$, so the signal received by the output layer is $w_i x_i$.
- Signals are aggregated as $\sum_{i=1}^{d} w_i x_i = \mathbf{w}^T \mathbf{x}$.

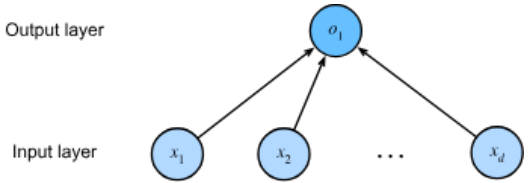


Figure: A linear model as a neural network.

Figure from https://d2l.ai/chapter_linear-regression/linear-regression.html

Linear model as a neural network

- Input: $\mathbf{x} = [x_1, \ldots, x_d]$ with $x_i \in \mathbb{R}$.
- Each neuron i has a weight $w_i \in \mathbb{R}$, so the signal received by the output layer is $w_i x_i$.
- Signals are aggregated as $\sum_{i=1}^{d} w_i x_i = \mathbf{w}^T \mathbf{x}$.

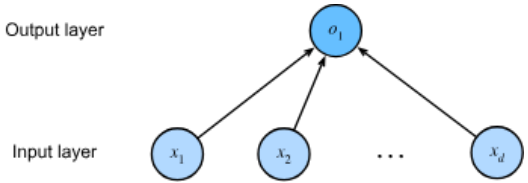


Figure: A linear model as a neural network.

Figure from https://d2l.ai/chapter_linear-regression/linear-regression.html

Activation Function

- Before passing signals to the next layer, a neuron may apply an **activation function** to the received signals.
- The output neuron o_1 processes the received signal $\mathbf{x}^T\mathbf{w}$, adds a bias term, and applies an activation function $\sigma : \mathbb{R} \to \mathbb{R}$, producing the final output:

$$\sigma(\mathbf{x}^T\mathbf{w} + b).$$

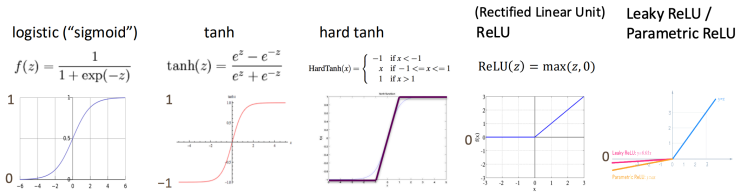


Figure: Examples of activation functions

Figure is adapted from Stanford cs224n

2-Layer (One-Hidden Layer) Neural Network

- A two-layer neural network.
- The 0th layer is the input layer with $d^{[0]}$ neurons.
- The 1st layer is the hidden layer with $d^{[1]}$ neurons.
- The 2nd layer is the output layer with $d^{[2]}$ neurons.

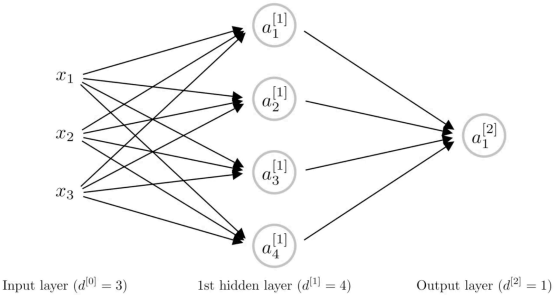


Figure: Example of a two-layer neural network.

Figure adapted from Zhonglei Wang

Outputs of the First Layer

- With a single data point $(\mathbf{x}, y)$, the pre-activation of the first neuron in the 1st layer is

$$z_1^{[1]} = \sum_{i=1}^{d^{[0]}} x_i w_{1i}^{[1]} + b_1^{[1]} = \mathbf{x}^T \mathbf{w}_1^{[1]} + b_1^{[1]},$$

  where $\mathbf{w}_1^{[1]} = (w_{11}^{[1]}, \ldots, w_{1d^{[0]}}^{[1]})$ is a weight vector.
- The output of the first neuron after activation:

$$a_1^{[1]} = \sigma(z_1^{[1]}).$$

- The first layer outputs a vector:

$$\mathbf{a}^{[1]} = (a_1^{[1]}, \ldots, a_{d^{[1]}}^{[1]}).$$

- In matrix form:

$$\mathbf{a}^{[1]} = \sigma\left(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}\right),$$

  where $\mathbf{W}^{[1]} = [\mathbf{w}_1^{[1]}, \ldots, \mathbf{w}_{d^{[1]}}^{[1]}] \in \mathbb{R}^{d^{[1]} \times d^{[0]}}$ and $\mathbf{b}^{[1]} = [b_1^{[1]}, \cdots, b_{d^{[1]}}]$. (Here, σ is applied element-wise.)

Outputs of the Final Layer

- Using matrix notation, with weight matrix $\mathbf{W}^{[2]} \in \mathbb{R}^{d^{[2]} \times d^{[1]}}$ and bias vector $\mathbf{b}^{[2]}$, the output of the final layer is

$$\mathbf{a}^{[2]} = \sigma\left(\mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]}\right).$$

- Define the function:

$$f_{\mathbf{W}^{[1]},\mathbf{b}^{[1]},\mathbf{W}^{[2]},\mathbf{b}^{[2]}}(\mathbf{x}) = \mathbf{a}^{[2]}.$$

Least Squares Regression with a 2-Layer Neural Network

- For each data point $\mathbf{z}_i = (\mathbf{x}_i, y_i)$, define the loss:

$$\ell_{\mathbf{W}^{[1]},\mathbf{b}^{[1]},\mathbf{W}^{[2]},\mathbf{b}^{[2]}}(\mathbf{z}_i) = \left(y_i - f_{\mathbf{W}^{[1]},\mathbf{b}^{[1]},\mathbf{W}^{[2]},\mathbf{b}^{[2]}}(\mathbf{x}_i)\right)^2.$$

- Let θ represent all parameters:

$$\theta = [\mathrm{Vec}(\mathbf{W}^{[1]}), \mathbf{b}^{[1]}, \mathrm{Vec}(\mathbf{W}^{[2]}), \mathbf{b}^{[2]}].$$

- Rewrite the loss function as

$$\ell_\theta(\mathbf{z}_i) = \left(y_i - f_\theta(\mathbf{x}_i)\right)^2.$$

- The final objective function is

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell_\theta(\mathbf{z}_i).$$

Deep Neural Networks (DNN): Diagram for 3-Layer NN

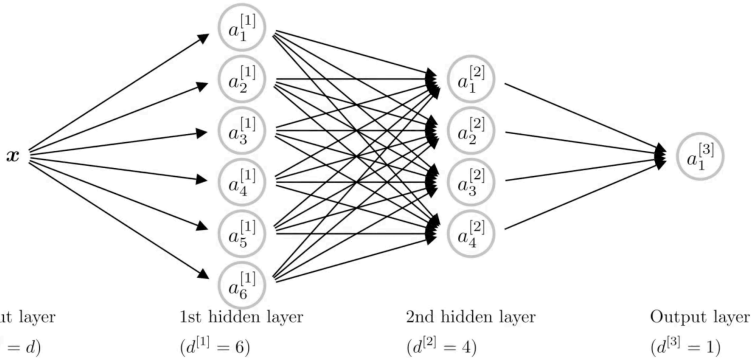


Figure from Zhonglei Wang

Deep Neural Networks: Mathematics

- A DNN with depth L (i.e., L layers) is represented as

$$f_\theta(\mathbf{x}) = \mathbf{W}^{[L]}\sigma(\mathbf{W}^{[L-1]}\sigma(\cdots\sigma(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]})) + \mathbf{b}^{[L-1]})) + \mathbf{b}^{[L]}.$$

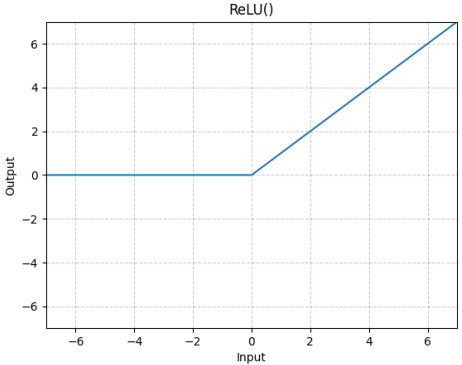
- Here, θ is a vector containing all parameters:

$$\theta = \{\mathbf{W}^{[l]}, \mathbf{b}^{[l]}\}_{l=1}^{L}.$$

- The final layer is typically fully connected to one neuron or may use a *special* activation function such as SoftMax.
- Given θ, computing the final output is known as forward propagation.

ReLU Activation Function (important)

- A Rectified Linear Units (ReLU) activation function has the form $\sigma(t) = \max\{0, t\}$.
- Motivation: eliminate the negative signal.



Building a Deep Fully Connected Neural Network in PyTorch

Demonstration: Constructing a 3-layer fully connected neural network using pytorch (3-layer_FCNN.ipynb).
Tutorial: https://www.bilibili.com/video/BV1wuRPYdEPD/?spm_id_from=333.1387.homepage.video_card.click

Recall: Gradient and derative

- For a multivariate function $f(\mathbf{x})$ with $\mathbf{x} = [x_1, \cdots, x_d]^T \in \mathbb{R}^d$ (column vector), the gradient of f is a column vector defined as

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}) \\ \vdots \\ \frac{\partial f}{\partial x_d}(\mathbf{x}) \end{bmatrix} \in \mathbb{R}^d.$$

- Its derivative (a row vector in this case) is given by

$$\mathbf{D}f = \nabla^T f = \begin{bmatrix} \frac{\partial f}{\partial x_1}, \cdots, \frac{\partial f}{\partial x_d} \end{bmatrix}.$$

Recall: Jacobian Matrix of cector-valued Functions

- For a vector-valued multivariate function $\mathbf{f}(\mathbf{x}) = [f_1(\mathbf{x}), \ldots, f_n(\mathbf{x})]^T \in \mathbb{R}^n$, where $\mathbf{x} = [x_1, \ldots, x_d]^T \in \mathbb{R}^d$ is a column vector, the Jacobian matrix of $\mathbf{f}$ is defined as:

$$\mathbf{Df}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}) & \cdots & \frac{\partial f_1}{\partial x_d}(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1}(\mathbf{x}) & \cdots & \frac{\partial f_n}{\partial x_d}(\mathbf{x}) \end{bmatrix} \in \mathbb{R}^{n \times d}.$$

- When $n = 1$, the Jacobian reduces to the derivative vector.

Recall: Chain rules (calculus)

- For two univariate function g, h, the chain rule says

$$(g \circ h)'(x) = g'(h(x)) \cdot h'(x)$$

- For two vector-valued multivariate functions, $\mathbf{g}, \mathbf{h}$, it holds

$$\mathbf{D}(\mathbf{g} \circ \mathbf{h})((\mathbf{x})) = \mathbf{D}\mathbf{g}(\mathbf{h}(\mathbf{x}))\ \mathbf{D}(\mathbf{h}(\mathbf{x})))$$

- Exercise: $f(\mathbf{w}) = \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2$

Gradient Descent for Least Squares

- Recall the linear regression loss:

$$\ell_{\mathbf{w},b}(\mathbf{z}_i) = (y_i - \mathbf{w}^T\mathbf{x}_i - b)^2.$$

- Its gradients are

$$\frac{\partial \ell_{\mathbf{w},b}(\mathbf{z}_i)}{\partial \mathbf{w}} = 2(\mathbf{w}^T\mathbf{x}_i + b - y_i)\mathbf{x}_i,$$
$$\frac{\partial \ell_{\mathbf{w},b}(\mathbf{z}_i)}{\partial b} = 2(\mathbf{w}^T\mathbf{x}_i + b - y_i).$$

- At each iteration, with step size η, update:

$$\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}} - \frac{2\eta}{n}\sum_{i=1}^{n}(\mathbf{x}_i^T\mathbf{w}^{\text{old}} + b^{\text{old}} - y_i)\mathbf{x}_i, \tag{1}$$

$$b^{\text{new}} = b^{\text{old}} - \frac{2\eta}{n}\sum_{i=1}^{n}(\mathbf{x}_i^T\mathbf{w}^{\text{old}} + b^{\text{old}} - y_i). \tag{2}$$

Stochastic gradient descent

- At each iteration, choose one data point uniformly at random from $\{\mathbf{z}_i\}_{i=1}^n$.
- At the k-th iteration, select i_k such that $\mathrm{Prob}[i_k = j] = 1/n$ for any $j = 1, \cdots, n$.
- The stochastic gradient descent is defined iterative as

$$\theta_{k+1} = \theta_k - \nabla_\theta \ell_{\theta_k}(\mathbf{z}_{i_k}),$$

with some random initialization θ_0.

Minibatch Stochastic gradient descent

- At each iteration, sample b ($b<n$) indices uniformly at random from $\{1, \cdots, n\}$ and let $\mathcal{B}$ be collection of all chosen indices ($|\mathcal{B}| = b$).
- With step size η, update:

$$\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}} - \frac{2\eta}{b} \sum_{i \in \mathcal{B}} (\mathbf{x}_i^T \mathbf{w}^{\text{old}} + b^{\text{old}} - y_i) \mathbf{x}_i,$$

$$b^{\text{new}} = b^{\text{old}} - \frac{2\eta}{b} \sum_{i \in \mathcal{B}} (\mathbf{x}_i^T \mathbf{w}^{\text{old}} + b^{\text{old}} - y_i).$$

Computational complexity

- OLS: $O(nd^2 + d^3)$
- Let K be the total number of iterations.
- GD: $O(K \cdot n \cdot d)$
- Mini-batch SGD: $K \cdot b \cdot d$
- SGD: $O(Kd)$

Train a model in python using SGD (Demonstration)

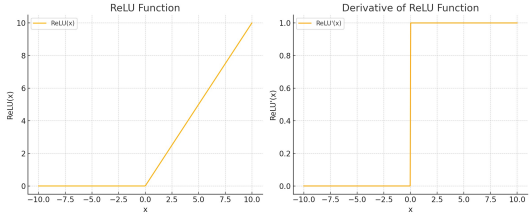
Linear_regression_SGD.ipynb

Train a model using pytorch (Demonstration)

linear_regression_as_nn.ipynb
linear_data_spliting.ipynb

Gradient for ReLU

- Recall ReLU activation function $\sigma(t) = \max\{0, t\}$.
- ReLU is not differentiable at $t = 0$.
- But it has a sub-derivative (for convex functions) at $t = 0$, which is a set $[0, 1]$.
- In practice, we may just take $\sigma'(0) = 0$.



Chain Rule for element-wise activation functions

- Consider $\mathbf{f}_{\mathbf{W},b}(\mathbf{x}) = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) =: \sigma \circ \mathbf{h}(\mathbf{W}, \mathbf{b})$. Here $\mathbf{h}(\mathbf{W}, \mathbf{b}) = \mathbf{W}\mathbf{x} + \mathbf{b} \in \mathbb{R}^{d^{[1]}}$ is a linear vector-valued function.
- $\mathbf{D}\sigma(\mathbf{h}) = \mathrm{Diag}[\sigma'(h_1), \cdots, \sigma'(h_{d^{[1]}})] :=$

$$\begin{bmatrix} \sigma'(h_1), & 0 & \cdots & 0 \\ 0 & \sigma'(h_2), & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & \sigma'(h_{d^{[1]}}) \end{bmatrix}$$

- $\mathbf{D}\mathbf{f}(\mathbf{W}_1, \mathbf{b}) = \mathbf{D}\sigma(\mathbf{h}) \quad \mathbf{D}\mathbf{h}(\mathbf{W}, \mathbf{b}) =$

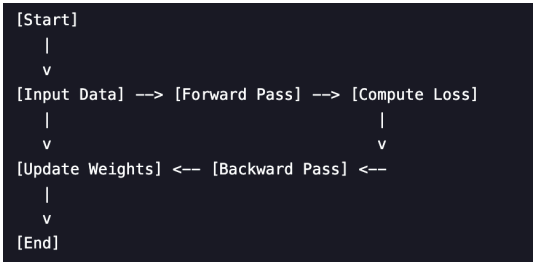
$$\begin{bmatrix} \sigma'(h_1), & 0 & \cdots & 0 \\ 0 & \sigma'(h_2), & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & \sigma'(h_{d^{[1]}}) \end{bmatrix} \begin{bmatrix} \mathbf{x}, 1 \\ \mathbf{x}, 1 \\ \vdots \\ \mathbf{x}, 1 \end{bmatrix}$$

Takeaways

- What if the model is more complex than a linear model?
- Train a Word2Vec model using SGD.
- Other variants of SGD (using momentum): Adam, Adam-W, Nesterov acceleration etc.
- What's next? (Backpropagation (autograd), classification problems)

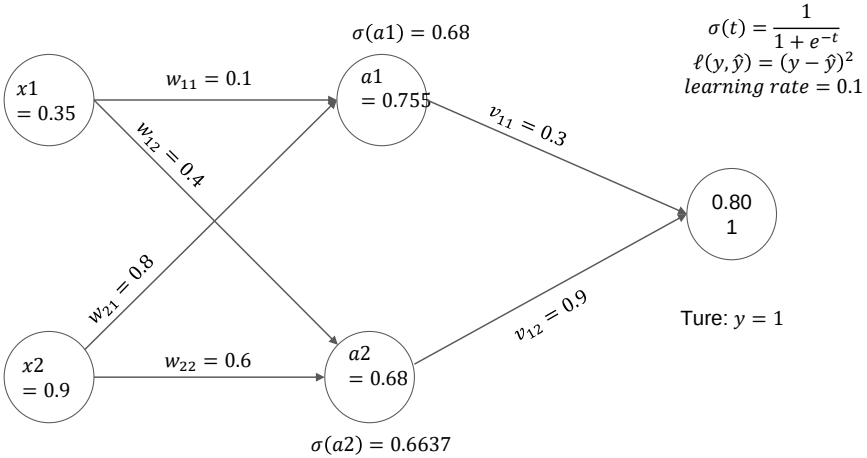
- In general, it is not easy to find a close-form representation of the gradient.

Update weights using back propagation



- Forward propagation: $\mathbf{x}_i \Rightarrow$ hidden layers outputs $\Rightarrow f_\theta(\mathbf{x}_i) \Rightarrow \ell_\theta(\mathbf{z}_i)$
- Back propagation: $\ell_\theta(\mathbf{z}_i) \Rightarrow$ layer-wise gradient $\Rightarrow \theta^{\text{new}}$

Example

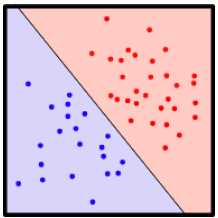

$\sigma(a1) = 0.68$
$\sigma(t) = \frac{1}{1+e^{-t}}$
$\ell(y, \hat{y}) = (y - \hat{y})^2$
$learning\ rate = 0.1$
$x1 = 0.35$
$w_{11} = 0.1$
$a1 = 0.755$
$v_{11} = 0.3$
$w_{12} = 0.4$
0.80
1
$w_{21} = 0.8$
$v_{12} = 0.9$
Ture: $y = 1$
$x2 = 0.9$
$w_{22} = 0.6$
$a2 = 0.68$
$\sigma(a2) = 0.6637$

Takeaway

- What is the computational complexity of forward propagation and backpropagation?
- Which one is faster? How much faster?

Binary classification

- $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ with $y_i \in \{0, 1\}$
- Objective: given a new feature $\mathbf{x}$, predict its label ($\mathbb{P}[y = 1 \text{ or } 0|x]$).



Logistic loss

- For a function (a.k.a., a classifier for classification problems) $f_\theta(\mathbf{x}) \in [0,1]$ parameterized by θ, the logistic loss at a data point $\mathbf{z} = (\mathbf{x}, y)$ is defined as

$$\ell_\theta(\mathbf{z}) = -y \log f_\theta(\mathbf{x}) - (1-y)\log(1 - f_\theta(\mathbf{x})).$$

- Empirical loss: $L(\theta) = \frac{1}{n}\sum_{i=1}^n \ell_\theta(\mathbf{z}_i)$.
- Rewrite $\mathbf{y} = (y^{(1)}, y^{(2)}) = (y, 1-y)$ and $\mathbf{f}_\theta(\mathbf{x}) = (f_\theta^{(1)}(\mathbf{x}), f_\theta^{(2)}(\mathbf{x})) = (f_\theta(\mathbf{x}), 1 - f_\theta(\mathbf{x}))$.
- Then, the loss function becomes

$$\ell_\theta(\mathbf{z}) = \sum_{j=1}^{2} y^{(j)} \log f_\theta^{(j)}(\mathbf{x})$$

- $\widetilde{\mathbf{y}}$ is called the one-hot encoding of y: $\widetilde{\mathbf{y}} = [1, 0]$ if $y = 1$.

Multiclass classification

- What if we have K classes and the label $y \in \{1, \cdots, K\}$?
- One hot encoding $\mathbf{y} = (0, \cdots, 0, 1, 0 \cdots, 0)$. (Only the k-th element is 1 and the rest are 0s if $y = k$.)
- Classifier: $\mathbf{f}_\theta(\mathbf{x}) = (f_\theta^{(1)}(\mathbf{x}), f_\theta^{(2)}(\mathbf{x}), \cdots, f_\theta^{(K)}(\mathbf{x})) \in [0, 1]^K$.

Cross entropy loss

- Recall the logistic loss when $K = 2$:

$$\ell_\theta(\mathbf{z}) = \sum_{j=1}^{2} y^{(j)} \log f_\theta^{(j)}(\mathbf{x}).$$

- For K-class classification problems, we extend the logistic loss to the cross entropy loss:

$$\ell_\theta^{\mathrm{CE}}(\mathbf{z}) = \sum_{j=1}^{K} y^{(j)} \log f_\theta^{(j)}(\mathbf{x}).$$

Neural networks for classification, I

- For classification problems, the final layer should contain K neurons.

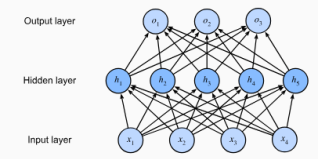


Figure: A neural network for a classification problem with 3-classes.

Figure is from *Dive into deep learning*

Neural networks for classification, II

- Remember that we require $f_\theta^{(i)} \in [0, 1]$.
- Use SoftMax function as an activation function of the last layer and we got

$$\mathbf{f}_\theta(\mathbf{x}) = \mathrm{SoftMax}\left(\mathbf{W}^{[L]}\sigma(\mathbf{W}^{[L-1]}\sigma(\cdots\sigma(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]})) + \mathbf{b}^{[L-1]})\right).$$

- So the final output is a probability (a K-dimensional vector).
- $f_\theta^{(k)}(\mathbf{x}) \approx \mathbb{P}[y = k|\mathbf{x}]$. (How to test)?

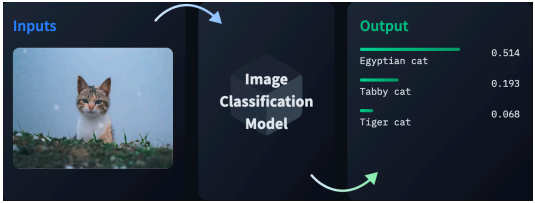


Figure is from Hugging face

Train

- The training procedure is to solve

$$\widehat{\theta} = \operatorname{argmin}_{\theta} \left\{ \frac{1}{n} \sum_{i=1}^{n} \ell_{\theta}^{\mathrm{CE}}(\mathbf{z}_i) \right\}.$$

- Use backpropagation

Mis-classification error and test

- For a probability $\mathbf{p} = (p_1, \cdots, p_K)$, let $\mathrm{OneHot}(\mathbf{p}) = [0, \cdots, 0, 1, 0, \cdots, 0]$ where k-th element is 1 if p_k is the largest among $\{p_1, \cdots, p_K\}$ (the rest are zeros).
- The mis-classification error is defined as

$$\mathrm{Prob}_{(\mathbf{x},y)\sim\mathbb{P}}[\mathbf{y} \neq \mathrm{OneHot}(\mathbf{f}_\theta(\mathbf{x}))].$$

- With test data $\{(\mathbf{x}_i^{\mathrm{test}}, y_i^{\mathrm{test}})\}_{i=1}^m$, the testing error is

$$\frac{1}{m}\sum_{i=1}^{m} \mathbf{1}_{[\mathbf{y}_i^{\mathrm{test}} \neq \mathrm{OneHot}(\mathbf{f}_\theta(\mathbf{x}_i^{\mathrm{test}}))]},$$

here $\mathbf{1}$ is an indicator function ($\mathbf{1}_A = 1$ if A happens, otherwise $\mathbf{1}_A = 0$).

Some questions

- Why do we use different loss functions in training and testing procedures? (because the indicator function is non-covex)
- Why do we use the logistic loss (or CE loss)? (Hint: Just like the least squares loss, consider the minimizer of $\mathbb{E}_{\mathbf{z}}[\ell_f(\mathbf{z})]$ and relate it to $\mathrm{Prob}[y = 1|\mathbf{x}]$).

Something advanced

- Other loss functions: e.g., Hinge loss, least squares loss for classification.
- Benchmarking datasets: MNIST, CIFAR-10, CIFAR-100, ImageNet

More advanced

- Train a classification model using pytorch yourself on the benchmarking datasets.
- Prevent overfitting?