Deep Learning

Muchang Bahng

June 16, 2023

Contents

There are two mainstream packages that impelement deep learning: Tensorflow and PyTorch. We will use PyTorch in here.

1 Multi-Layered Perceptron

First, we transform the inputs into the relevant features $\mathbf{x}_n \mapsto \boldsymbol{\phi}(\mathbf{x}_n) = \boldsymbol{\phi}_n$ and then, when we construct a generalized linear model, we assume that the conditional distribution $Y \mid X = x$ is in the canonical exponential family, with some natural parameter $\eta(x)$ and expected mean $\mu(x) = \mathbb{E}[Y \mid X = x]$. Then, to choose the link function g that related $g(\mu(x)) = x^T \beta$, we can set it to be the canonical link g that maps μ to η. That is,

$$g(\mu(x)) = x^T \beta = \eta(x)$$

such that the natural parameter is linearly dependent on the input. The inverse g^{-1} of the link function is called the **activation function**, which connects the expected mean to a linear function of x.

$$h_\beta(x) = g^{-1}(x^T \beta) = \mu(x) = \mathbb{E}[Y \mid X = x]$$

Now, note that for a classification problem, the decision boundary defined in the $\boldsymbol{\phi}$ feature space is linear, but it may not be linear in the input space $\mathcal{X}$. For example, consider the set of points in $\mathbb{R}^2$ with the corresponding class in Figure **??**. We transform the features to $\boldsymbol{\phi}(x_1, x_2) = x_1^2 + x_2^2$, which gives us a new space to work with. Fitting logistic regression onto this gives a linear decision boundary in the space $\boldsymbol{\phi}$, but the boundary is circular in $\mathcal{X} = \mathbb{R}^2$.

We would like to extend this model by making the basis functions $\boldsymbol{\phi}_n$ depend on the parameters $\mathbf{w}$ and then allow these parameters to be adjusted during training. There are many ways to construct parametric nonlinear basis functions and in fact, neural networks use basis functions that are of the form $\boldsymbol{\phi}(\mathbf{x}) = g^{-1}(\mathbf{x}^T \boldsymbol{\beta})$.

A neuron basically takes in a vector $\mathbf{x} \in \mathbb{R}^d$ and multiplies its corresponding weight by some vector $\boldsymbol{\omega}$, plus some bias term b. It is then sent into some nonlinear activation function $\sigma : \mathbb{R} \longrightarrow \mathbb{R}$. Letting the parameter be $\theta = (\boldsymbol{\omega}, b)$, we can think of a neuron as a function

$$h_\theta(\mathbf{x}) = f(\boldsymbol{\omega}^T \mathbf{x} + b)$$

A single neuron with the activation function as the step function

$$f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

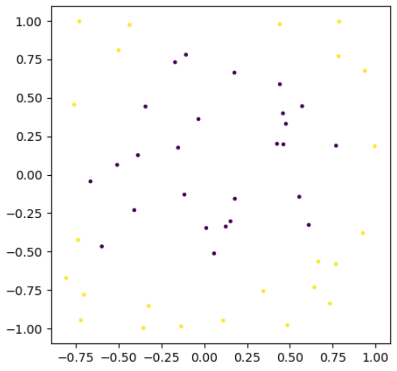
is simply the perceptron algorithm. It divides $\mathbb{R}^d$ using a hyperplane $\boldsymbol{\omega}^T \mathbf{x} + b = 0$ and linearly classifies all points on one side to value 1 and the other side to value 0. This is similar to a neuron, which takes in a value and outputs a "signal" if the function evaluated gets past a threshold. However, we would like to use smooth activation functions for this, so we would use different activations. Hence we have a neuron.

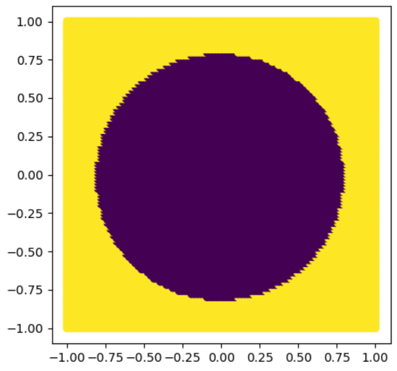
Definition 1.1 (Neuron)**.** A **neuron** is a function (visualized as a node) that takes in inputs $\mathbf{x}$ and outputs a value y calculated

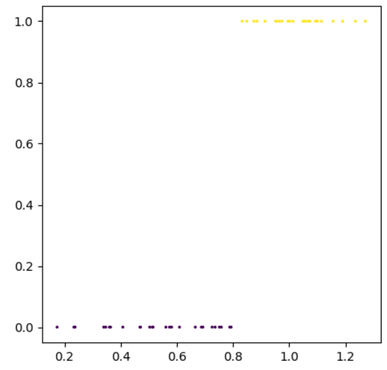
$$y = \sigma(\mathbf{w}^T x + b)$$

where σ is an activation function. Activation functions are usually simple functions with a range of $[0, 1]$ or $[-1, 1]$, and popular ones include:

1. the rectified linear unit
$$\text{ReLU}(z) = \max\{0, z\}$$
2. the sigmoid
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$


(a) Data in space $\mathcal{X} = \mathbb{R}^2$.


(b) Logistic fit to data in input space.


(c) Transformed data $\phi(\mathbf{x}) = ||\mathbf{x}||$.

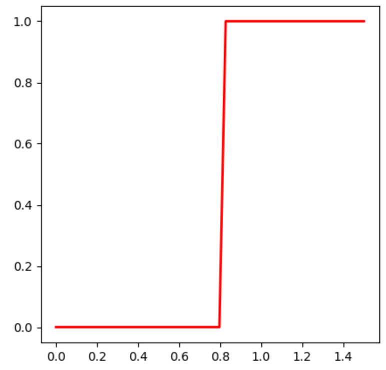

(d) Logistic fit in transformed space.

Figure 1: A nonlinear feature transformation ϕ will cause a nonlinear decision boundary when doing logistic regression.

3. the hyperbolic tangent
$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

A visual of a neuron is shown in Figure 2.

If there does not exist any arrow from a potential input $\mathbf{x}$ to an output y, then this means that $\mathbf{x}$ is not relevant in calculating y. However, we usually work with **fully-connected neural networks**, which means that every input is relevant to calculating every output, since we usually cannot make assumptions about which variables are relevant or not. We can stack multiple neurons such that one neuron passes its output as input into the next neuron, resulting in a more complex function. What we have seen just now is a 1-layer neural network.

Definition 1.2 (Multilayer Perceptron)**.** A L-layer MLP $\mathbf{h}_\theta : \mathbb{R}^D \longrightarrow \mathbb{R}^M$ is the function
$$h_\theta(\mathbf{x}) \coloneqq \boldsymbol{\sigma}^{[L]} \circ \mathbf{W}^{[L]} \circ \boldsymbol{\sigma}^{[L-1]} \circ \mathbf{W}^{[L-1]} \circ \cdots \circ \boldsymbol{\sigma}^{[1]} \circ \mathbf{W}^{[1]}(\mathbf{x})$$
where $\boldsymbol{\sigma}^{[l]} : \mathbb{R}^{N^{[l]}} \to \mathbb{R}^{N^{[l]}}$ is an activation function and $\mathbf{W}^{[l]} : \mathbb{R}^{N^{[l-1]}} \to \mathbb{R}^{N^{[l]}}$ is an affine map. We will use the following notation.

1. The inputs will be labeled $\mathbf{x} = \mathbf{a}^{[0]}$ which is in $\mathbb{R}^{N^{[0]}} = \mathbb{R}^D$.

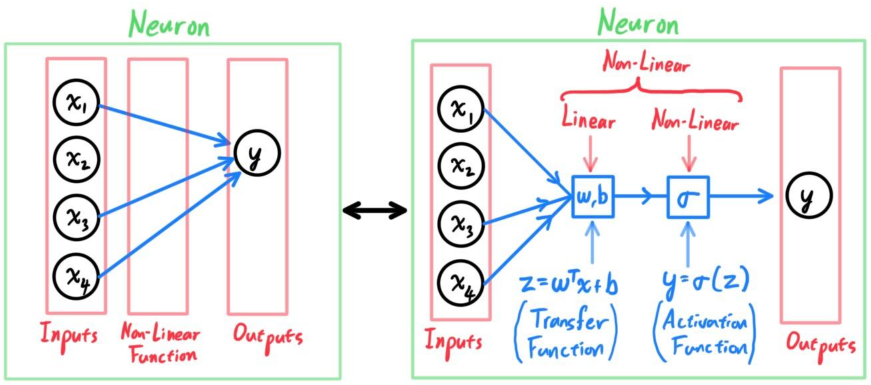
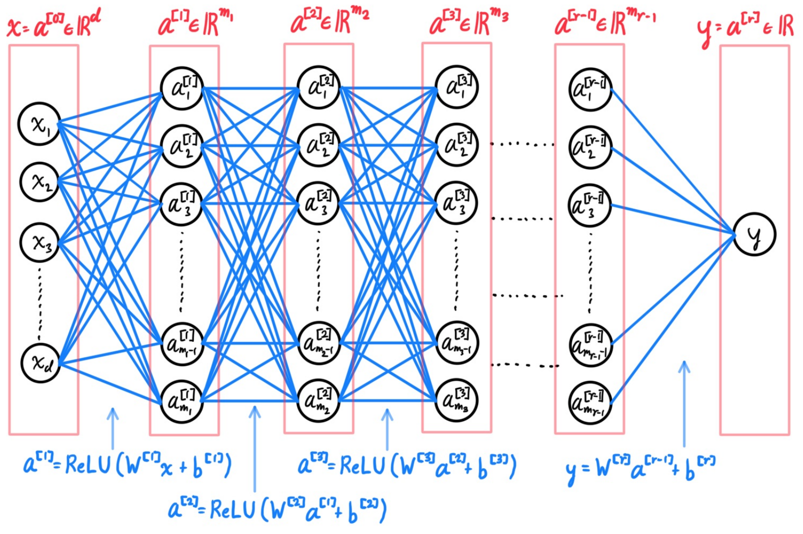


Figure 2: Diagram of a neuron, decomposed into its linear and nonlinear components.

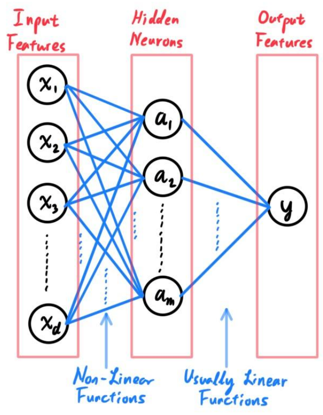
2. We map $\mathbf{a}^{[l]} \in \mathbb{R}^{N^{[l]}} \mapsto \mathbf{W}^{[l+1]}\mathbf{a}^{[l]} + \mathbf{b}^{[l+1]} = \mathbf{z}^{[l+1]} \in \mathbb{R}^{N^{[l+1]}}$, where z denotes a vector after an affine transformation.
3. We map $\mathbf{z}^{[l+1]} \in \mathbb{R}^{N^{[l+1]}} \mapsto \boldsymbol{\sigma}(\mathbf{z}^{[l+1]}) = \mathbf{a}^{[l+1]} \in \mathbb{R}^{N^{[l+1]}}$, where a denotes a vector after an activation function.
4. We keep doing this until we reach the second last layer with vector $\mathbf{a}^{[L-1]}$.
5. Now we want our last layer to be our predicted output. Based on our assumptions of the problem, we construct a generalized linear model with some inverse link function g. We perform one more affine transformation $\mathbf{a}^{[L-1]} \mapsto \mathbf{W}^{[L]}\mathbf{a}^{[L-1]} + \mathbf{b}^{[L]} = \mathbf{z}^{[L]}$, followed by the link function to get our prediction: $\mathbf{a}^{[L]} = \mathbf{g}(\mathbf{z}^{[L]}) = \mathbf{h}_{\boldsymbol{\theta}}(\mathbf{x}) \in \mathbb{R}^M$.

All the parameters of the neural net will be denoted $\boldsymbol{\theta}$. Ultimately, a neural net is really just a generalized linear model on an feature space with a ton of nonlinear preprocessing.


$x = a^{[0]} \in \mathbb{R}^d$
$a^{[1]} \in \mathbb{R}^{m_1}$
$a^{[2]} \in \mathbb{R}^{m_2}$
$a^{[3]} \in \mathbb{R}^{m_3}$
$a^{[r-1]} \in \mathbb{R}^{m_{r-1}}$
$y = a^{[r]} \in \mathbb{R}$
x_1
x_2
x_3
x_d
$a^{[1]}_1$
$a^{[1]}_2$
$a^{[1]}_3$
$a^{[1]}_{m_1-1}$
$a^{[1]}_{m_1}$
$a^{[2]}_1$
$a^{[2]}_2$
$a^{[2]}_3$
$a^{[2]}_{m_2-1}$
$a^{[2]}_{m_2}$
$a^{[3]}_1$
$a^{[3]}_2$
$a^{[3]}_3$
$a^{[3]}_{m_3-1}$
$a^{[3]}_{m_3}$
$a^{[r-1]}_1$
$a^{[r-1]}_2$
$a^{[r-1]}_3$
$a^{[r-1]}_{m_{r-1}-1}$
$a^{[r-1]}_{m_{r-1}}$
y
$a^{[1]} = \mathrm{ReLU}(W^{[1]}x + b^{[1]})$
$a^{[3]} = \mathrm{ReLU}(W^{[3]}a^{[2]} + b^{[3]})$
$y = W^{[r]}a^{[r-1]} + b^{[r]}$
$a^{[2]} = \mathrm{ReLU}(W^{[2]}a^{[1]} + b^{[2]})$

The last layer represents the key features that we are interested in, and in practice, if researchers want to predict a smaller dataset, they take a pretrained model on a related larger dataset and simply tune the final layer, since the second last layer most likely encodes all the relevant features.

Example 1.1. The **fully-connected 2-layer neural network** of d input features $\mathbf{x} \in \mathbb{R}^d$ and one scalar output $y \in \mathbb{R}$ can be visualized below. It has one **hidden layer** with m inputs values $a_1, \ldots, a_m$.



Conventionally, we account for every layer except for the final layer when talking about the number of layers in the neural net.

Note that each layer corresponds to how close a neuron is to the output. But really any neuron can be a function of any other neuron. For example, we can connect a neuron from layer 4 back to a neuron of layer 1. For now, we will consider networks that are restricted to a **feed-forward** architecture, in other words having no closed directed cycles.

1.1 Weight Space Symmetries

2 Network Training

Now we can essentially do the same regression analysis with neural nets. Assume that we have some neural net $h_{\boldsymbol{\theta}}$, and denote the set of all functions of this form to be $\mathcal{F} = \{h_{\boldsymbol{\theta}} \,:\, \boldsymbol{\theta} \in \mathbb{R}^M\}$, where M is the number of parameters in this model. Then, we will assume that $h_{\boldsymbol{\theta}}(X)$ approximates $\mathbb{E}[Y \mid X]$ in such a way that

$$Y = h_{\boldsymbol{\theta}}(X) + \epsilon, \;\; \epsilon \sim N(0, \sigma^2)$$

Then the distibution of $Y \mid X = x$ would have density

$$p(y \mid \mathbf{x}, \boldsymbol{\theta}) = N(y \mid h_{\boldsymbol{\theta}}(\mathbf{x}), \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(- \frac{\big(y - h_{\boldsymbol{\theta}}(\mathbf{x})\big)^2}{2\sigma^2}\right)$$

and taking the log likelihood of the dataset $\{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^N$ gives us

$$\ell(\boldsymbol{\theta}) = \frac{1}{2\sigma^2} \sum_{n=1}^N \big(y - h_{\boldsymbol{\theta}}(\mathbf{x})\big)^2 + \frac{N}{2} \ln \sigma^2 + \frac{N}{2} \ln(2\pi)$$

which clearly shows that we must minimize our sum of squares error.

Now remember that our neural net is really just a generalized linear model where we are learning the transformations in addition to the final parameter weights. Therefore, our outputs y may not be in $\mathbb{R}$ (such as softmax activation), which will give different loss functions, and so we should generalize our loss to be

$$E(\boldsymbol{\theta}) = \sum_{n=1}^N E_n\big[\mathbf{y}^{(n)}, h_{\boldsymbol{\theta}}(\mathbf{x}^{(n)})\big] = \sum_{n=1}^N E_n(\boldsymbol{\theta})$$

where E_n is the loss corresponding to the nth input-output pair.

2.1 Backpropagation

Backpropagation is not hard, but it is cumbersome notation-wise. What we really want to do is just compute a very long vector with all of its partials $\partial E / \partial \boldsymbol{\theta}$.

To compute $\frac{\partial E_n}{\partial w_{ji}^{[l]}}$, it would be natural to split it up into a portion where E_n is affected by the term before activation $\mathbf{z}^{[l]}$ and how that is affected by $w_{ji}^{[l]}$. The same goes for the bias terms.

$$\frac{\partial E_n}{\partial w_{ji}^{[l]}} = \underbrace{\frac{\partial E_n}{\partial \mathbf{z}^{[l]}}}_{1 \times N^{[l]}} \cdot \underbrace{\frac{\partial \mathbf{z}^{[l]}}{\partial w_{ji}^{[l]}}}_{N^{[l]} \times 1} \text{ and } \frac{\partial E_n}{\partial b_i^{[l]}} = \underbrace{\frac{\partial E_n}{\partial \mathbf{z}^{[l]}}}_{1 \times N^{[l]}} \cdot \underbrace{\frac{\partial \mathbf{z}^{[l]}}{\partial b_i^{[l]}}}_{N^{[l]} \times 1}$$

It helps to visualize that we are focusing on

$$\mathbf{h}_{\boldsymbol{\theta}}(\mathbf{x}) = g\big(\ldots \sigma(\underbrace{\mathbf{W}^{[l]} \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}}_{\mathbf{z}^{[l]}}) \ldots \big)$$

We can expand $\mathbf{z}^{[l]}$ to get

$$\mathbf{z}^{[l]} = \begin{pmatrix} w_{11}^{[l]} & \ldots & w_{1N^{[l-1]}}^{[l]} \\ \vdots & \ddots & \vdots \\ w_{N^{[l]}1}^{[l]} & \ldots & w_{N^{[l]}N^{[l-1]}}^{[l]} \end{pmatrix} \begin{pmatrix} a_1^{[l-1]} \\ \vdots \\ a_{N^{[l-1]}}^{[l-1]} \end{pmatrix} + \begin{pmatrix} b_1^{[l]} \\ \vdots \\ b_{N_{[l]}} \end{pmatrix}$$

$w_{ji}^{[l]}$ will only show up in the jth term of $\mathbf{z}^{[l]}$, and so the rest of the terms in $\frac{\partial \mathbf{z}^{[l]}}{\partial w_{ji}^{[l]}}$ will vanish. The same logic applies to $\frac{\partial \mathbf{z}^{[l]}}{\partial b_i^{[l]}}$, and so we really just have to compute

$$\frac{\partial E_n}{\partial w_{ji}^{[l]}} = \underbrace{\frac{\partial E_n}{\partial z_j^{[l]}}}_{1 \times 1} \cdot \underbrace{\frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}}}_{1 \times 1} = \delta_j^{[l]} \cdot \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}} \text{ and } \frac{\partial E_n}{\partial b_i^{[l]}} = \underbrace{\frac{\partial E_n}{\partial z_j^{[l]}}}_{1 \times 1} \cdot \underbrace{\frac{\partial z_j^{[l]}}{\partial b_i^{[l]}}}_{1 \times 1} = \delta_j^{[l]} \cdot \frac{\partial z_j^{[l]}}{\partial b_i^{[l]}}$$

where the $\delta_j^{[l]}$ is called the jth **error term** of layer l. If we look at the evaluated jth row,

$$z_j^{[l]} = w_{j1}^{[l]} a_1^{[l-1]} + \ldots w_{jN^{[l-1]}} a_{N^{[l-1]}}^{[l-1]} + b_j^{[l]}$$

We can clearly see that $\frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}} = a_i^{[l-1]}$ and $\frac{\partial z_j^{[l]}}{\partial b_i^{[l]}} = 1$, which means that our derivatives are now reduced to

$$\frac{\partial E_n}{\partial w_{ji}^{[l]}} = \delta_j^{[l]} a_i^{[l-1]}, \qquad \frac{\partial E_n}{\partial b_i^{[l]}} = \delta_j^{[l]}$$

What this means is that we must know the intermediate values $\mathbf{a}^{[l-1]}$ beforehand, which is possible since we would compute them using forward propagation and store them in memory. Now note that the partial derivatives at this point have been calculated without any consideration of a particular error function or activation function. To calculate $\boldsymbol{\delta}^{[L]}$, we can simply use the chain rule to get

$$\delta_j^{[L]} = \frac{\partial E_n}{\partial z_j^{[L]}} = \frac{\partial E_n}{\partial \mathbf{a}^{[L]}} \cdot \frac{\partial \mathbf{a}^{[L]}}{\partial z_j^{[L]}} = \sum_k \frac{\partial E_n}{\partial a_k^{[L]}} \cdot \frac{\partial a_k^{[L]}}{\partial z_j^{[L]}}$$

which can be rewritten in the matrix notation

$$\boldsymbol{\delta}^{[L]} = \left(\frac{\partial \mathbf{g}}{\partial \mathbf{z}^{[L]}} \right)^T \left(\frac{\partial E_n}{\partial \mathbf{a}^{[L]}} \right) = \underbrace{\begin{bmatrix} \frac{\partial g_1}{\partial z_1^{[L]}} & \ldots & \frac{\partial g_{N^{[L]}}}{\partial z_1^{[L]}} \\ \vdots & \ddots & \vdots \\ \frac{\partial g_1}{\partial z_{N^{[L]}}^{[L]}} & \ldots & \frac{\partial g_{N^{[L]}}}{\partial z_{N^{[L]}}^{[L]}} \end{bmatrix}}_{N^{[L]} \times N^{[L]}} \begin{bmatrix} \frac{\partial E_n}{\partial a_1^{[L]}} \\ \vdots \\ \frac{\partial E_n}{\partial a_{N^{[L]}}^{[L]}} \end{bmatrix}$$

Note that as soon as we make a model assumption on the form of the conditional distribution $Y \mid X = x$ (e.g. it is Gaussian), with it being in the exponential family, we immediately get two things: the loss function E_n (e.g. sum of squares loss), and the canonical link function $\mathbf{g}$

1. If we assume that $Y \mid X = x$ is Gaussian in a regression (of scalar output) setting, then our canonical link would be $g(x) = x$, which gives the sum of squares loss function. Note that since the output is a real-valued scalar, $\mathbf{a}^{[L]}$ will be a scalar (i.e. the final layer is one node, $N^{[L]} = 1$).

$$E_n = \frac{1}{2}(y^{(n)} - a^{[L]})^2$$

To calculate $\boldsymbol{\delta}^{[L]}$, we can simply use the chain rule to get

$$\delta^{[L]} = \frac{\partial E_n}{\partial z^{[L]}} = \frac{\partial E_n}{\partial a^{[L]}} \cdot \frac{\partial a^{[L]}}{\partial z^{[L]}} = a^{[L]} - y^{(n)}$$

2. For classification (of M classes), we would use the softmax activation function (with its derivative next to it for convenience)

$$\mathbf{g}(\mathbf{z}) = \mathbf{g}\Bigg(\begin{bmatrix} z_1 \\ \vdots \\ z_M \end{bmatrix} \Bigg) = \begin{bmatrix} e^{z_1} / \sum_k e^{z_k} \\ \vdots \\ e^{z_M} / \sum_k e^{z_k} \end{bmatrix}, \quad \frac{\partial g_k}{\partial z_j} = \begin{cases} g_j(1-g_j) & \text{if } k = j \\ -g_j g_k & \text{if } k \neq j \end{cases}$$

which gives the cross entropy error

$$E_n = -\mathbf{y}^{(n)} \cdot \ln\big(\mathbf{h}_{\boldsymbol{\theta}}(\mathbf{x}^{(n)})\big) = -\sum_i y_i^{(n)} \ln(a_i^{[L]})$$

where the $\mathbf{y}$ has been one-hot encoded into a standard unit vector in $\mathbb{R}^M$. To calculate $\boldsymbol{\delta}^{[L]}$, we can again use the chain rule again

$$\begin{aligned}
\delta_j^{[L]} &= \sum_k \frac{\partial E_n}{\partial a_k^{[L]}} \cdot \frac{\partial a_k^{[L]}}{\partial z_j^{[L]}} \\
&= -\sum_k \frac{y_k^{(n)}}{a_k^{[L]}} \cdot \frac{\partial a_k^{[L]}}{\partial z_j^{[L]}} \\
&= \bigg(-\sum_{k \neq j} \frac{y_k^{(n)}}{a_k^{[L]}} \cdot \frac{\partial a_k^{[L]}}{\partial z_j^{[L]}} \bigg) - \frac{y_j^{(n)}}{a_j^{[L]}} \cdot \frac{a_j^{[L]}}{\partial z_j^{[L]}} \\
&= \bigg(-\sum_{k \neq j} \frac{y_k^{(n)}}{a_k^{[L]}} \cdot -a_k^{[L]} a_j^{[L]} \bigg) - \frac{y_j^{(n)}}{a_j^{[L]}} \cdot a_j^{[L]}(1 - a_j^{[L]}) \\
&= a_j^{[L]} \underbrace{\sum_k y_k^{(n)}}_{1} - y_j^{(n)} = a_j^{[L]} - y_j^{(n)}
\end{aligned}$$

giving us

$$\boldsymbol{\delta}^{[L]} = \mathbf{a}_j^{[L]} - \mathbf{y}^{[L]}$$

Now that we have found the error for the last layer, we can continue for the hidden layers. We can again expand by chain rule that

$$\delta_j^{[l]} = \frac{\partial E_n}{\partial z_j^{[l]}} = \frac{\partial E_n}{\partial \mathbf{z}^{[l+1]}} \cdot \frac{\partial \mathbf{z}^{[l+1]}}{\partial z_j^{[l]}} = \sum_{k=1}^{N^{[l+1]}} \frac{\partial E_n}{\partial z_k^{[l+1]}} \cdot \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = \sum_{k=1}^{N^{[l+1]}} \delta_k^{[l+1]} \cdot \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}}$$

By going backwards from the last layer, we should already have the values of $\delta_k^{[l+1]}$, and to compute the second partial, we recall the way a was calculated

$$z_k^{[l+1]} = b_k^{[l+1]} + \sum_{j=1}^{N^{[l]}} w_{kj}^{[l+1]} \sigma(z_j^{[l]}) \implies \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = w_{kj}^{[l+1]} \cdot \sigma'(z_j^{[l]})$$

Now this is where the "back" in backpropagation comes from. Plugging this into the equation yields a final equation for the error term in hidden layers, called the **backpropagation formula**:

$$\delta_j^{[l]} = \sigma'(z_j^{[l]}) \sum_{k=1}^{N^{[l+1]}} \delta_k^{[l+1]} \cdot w_{kj}^{[l+1]}$$

which gives the matrix form

$$\boldsymbol{\delta}^{[l]} = \boldsymbol{\sigma}'(\mathbf{z}^{[l]}) \odot (\mathbf{W}^{[l+1]})^T \boldsymbol{\delta}^{[l+1]} = \begin{bmatrix} \sigma'(z_1^{[l]}) \\ \vdots \\ \sigma'(z_{N^{[L]}}^{[l]}) \end{bmatrix} \odot \begin{bmatrix} w_{11}^{[l+1]} & \dots & w_{N^{[l+1]}1}^{[l+1]} \\ \vdots & \ddots & \vdots \\ w_{1N^{[l]}}^{[l+1]} & \dots & w_{N^{[l+1]}N^{[l]}}^{[l+1]} \end{bmatrix} \begin{bmatrix} \delta_1^{[l+1]} \\ \vdots \\ \delta_{N^{[l+1]}}^{[l+1]} \end{bmatrix}$$

and putting it all together, the partial derivative of the error function E_n with respect to the weight in the hidden layers for $1 \leq l < L$ is

$$\frac{\partial E_n}{\partial w_{ji}^{[l]}} = a_i^{[l-1]} \sigma'(z_j^{[l]}) \sum_{k=1}^{N^{[l+1]}} \delta_k^{[l+1]} \cdot w_{kj}^{[l+1]}$$

2.1.1 Summary

Therefore, let us summarize what a MLP does:

1. *Initialization*: We initialize all the parameters to be
$$\boldsymbol{\theta} = (\mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \mathbf{W}^{[2]}, \dots, \mathbf{W}^{[L]}, \mathbf{b}^{[L]})$$
2. *Choose Batch*: We choose an arbitrary data point $(\mathbf{x}^{(n)}, \mathbf{y}^{(n)})$, an minibatch, or the entire batch to compute the gradients on.
3. *Forward Propagation*: Apply input vector $\mathbf{x}^{(n)}$ and use forward propagation to compute the values of all the hidden and activation units
$$\mathbf{a}^{[0]} = \mathbf{x}^{(n)}, \mathbf{z}^{[1]}, \mathbf{a}^{[1]}, \dots, \mathbf{z}^{[L]}, \mathbf{a}^{[L]} = h_{\boldsymbol{\theta}}(\mathbf{x}^{(n)})$$
4. *Back Propagation*:
   (a) Evaluate the $\boldsymbol{\delta}^{[l]}$'s starting from the back with the formula
$$\boldsymbol{\delta}^{[L]} = \left(\frac{\partial \mathbf{g}}{\partial \mathbf{z}^{[L]}}\right)^T \left(\frac{\partial E_n}{\partial \mathbf{a}^{[L]}}\right)$$
$$\boldsymbol{\delta}^{[l]} = \boldsymbol{\sigma}'(\mathbf{z}^{[l]}) \odot (\mathbf{W}^{[l+1]})^T \boldsymbol{\delta}^{[l+1]} \qquad l = 1, \dots, L-1$$
   where $\frac{\partial \mathbf{g}}{\partial \mathbf{z}^{[L]}}$ can be found by taking the derivative of the known link function, and the rest of the terms are found by forward propagation (these are all functions which have been fixed in value by inputting $\mathbf{x}^{(n)}$).
   (b) Calculate the derivatives of the error as
$$\frac{\partial E_n}{\partial \mathbf{W}^{[l]}} = \boldsymbol{\delta}^{[l]} (\mathbf{a}^{[l-1]})^T, \qquad \frac{\partial E_n}{\partial \mathbf{b}^{[l]}} = \boldsymbol{\delta}^{[l]}$$
5. *Gradient Descent*: Subtract the derivatives with step size α. That is, for $l = 1, \dots, L$,
$$\mathbf{W}^{[l]} = \mathbf{W}^{[l]} - \alpha \frac{\partial E_n}{\partial \mathbf{W}^{[l]}}, \qquad \mathbf{b}^{[l]} = \mathbf{b}^{[l]} - \alpha \frac{\partial E_n}{\partial \mathbf{b}^{[l]}}$$

Note that many operations can run in parallel, so the ordering of these steps may be swapped.

2.2 Implementation of Neural Net in Python

Now let us implement a neural network with batch gradient descent in Python from scratch, using only Numpy. We will train on the MNIST dataset where the train and test sets can be gotten using the following commands.

```
import numpy as np
import torchvision.datasets as datasets

train_set = datasets.MNIST('./data', train=True, download=True)
test_set = datasets.MNIST('./data', train=False, download=True)

# Check the lengths of train sets and test sets
assert len(train_set) == 60000 and len(test_set) == 10000
```

Now we want to take each 28×28 image and flatten it out to a 784 vector.

```
X_train = np.array([picture.numpy().reshape(-1) for picture in train_set.data]).T / 255.
Y_train = train_set.targets.numpy()
X_test = np.array([picture.numpy().reshape(-1) for picture in test_set.data]).T / 255.
Y_test = test_set.targets.numpy()

# Check shapes
assert X_train.shape == (784, 60000) and Y_train.shape == (60000, )
```

Here are some helper functions that we will need.

```
def initialize_params():
    W1 = np.random.uniform(-1, 1, size=(10, 784))
    b1 = np.random.uniform(-1, 1, size=(10, 1))
    W2 = np.random.uniform(-1, 1, size=(10, 10))
    b2 = np.random.uniform(-1, 1, size=(10, 1))
    return W1, b1, W2, b2

def oneHot(Y):
    # Y is 60000
    oneHotY = np.zeros((10, Y.size))
    oneHotY[Y, np.arange(Y.size)] = 1
    return oneHotY # 10x60000

def ReLU(Z):
    return np.maximum(0, Z)

def ReLU_d(Z):
    return Z > 0

def softMax(X:np.array):
    x_max = np.max(X, axis=0)
    X = X - x_max
    return np.exp(X) / np.sum(np.exp(X), axis=0)

def softMax_d(X:np.array):
    sm = softMax(X)
    return - (np.diag(sm.sum(axis=1)) - np.matmul(sm, np.transpose(sm))) / X.shape[1]
```

Now we implement the forward propagation and back propagation.

```
def forwardProp(W1, b1, W2, b2, X):
    # Z1 12x1, W1 12x784 , X 784x1, b1 12x1
    Z1 = np.matmul(W1, X) + b1
    # A1 12x60000
    A1 = ReLU(Z1)

    # Z2 10x1, W2 10x12, A1 12x60000, b2 10x60000
    Z2 = np.matmul(W2, A1) + b2
    # A2 10x60000
    A2 = softMax(Z2)
    return Z1, A1, Z2, A2

def backProp(Z1, A1, Z2, A2, W1, W2, X, Y):
    N = Y.size
    oneHotY = oneHot(Y)

    # 10x1 = 10x10 10x1
    error2 = A2 - oneHotY
    # error2 = np.matmul(np.transpose(softMax_d(Z2)), A2 - oneHotY)
    # 10x12 = 10x1 1x12
    dW2 = 1/N * np.matmul(error2, A1.T)
    # 10x1
    dB2 = 1/N * error2.sum(axis=1)

    # 12x1 = 12x1 .* 12x10 10x1
    error1 = np.vectorize(ReLU_d)(Z1) * np.matmul(W2.T, error2)
    # 12x784 = 12x1 1x784
    dW1 = 1/N * np.matmul(error1, X.T)
    # 12x1
    dB1 = 1/N * error1.sum(axis=1)

    return dW1, dB1, dW2, dB2
```

We now build the batch gradient descent algorithm.

```
def get_predictions(A2):
    return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size

def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1.reshape(-1, 1)
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2.reshape(-1, 1)
    return W1, b1, W2, b2

def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = initialize_params()
    for i in range(iterations):
        Z1, A1, Z2, A2 = forwardProp(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backProp(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))
    return W1, b1, W2, b2

# Run it
W1, b1, W2, b2 = gradient_descent(X_train, Y_train, 0.1, 500)
```

Which gives the following output, ultimately yielding a 77% accuracy rate in detecting handwritten digits.

```
Iteration:  0
[4 7 4 ... 4 7 0] [5 0 4 ... 5 6 8]
0.10175
Iteration:  10
[8 0 4 ... 4 6 0] [5 0 4 ... 5 6 8]
0.20715
Iteration:  20
[8 0 4 ... 4 6 0] [5 0 4 ... 5 6 8]
0.2333
Iteration:  30
[9 0 4 ... 4 6 0] [5 0 4 ... 5 6 8]
0.2579166666666667
...
...
...
Iteration:  470
[3 0 9 ... 5 6 0] [5 0 4 ... 5 6 8]
0.7645
Iteration:  480
[3 0 9 ... 5 6 0] [5 0 4 ... 5 6 8]
0.7646833333333334
Iteration:  490
[3 0 9 ... 5 6 0] [5 0 4 ... 5 6 8]
0.7728166666666667
```

2.3 Normalization Layers

Normalization layers are a type of layer commonly used in neural networks to improve the stability and efficiency of training. They help address issues related to input data distributions and internal covariate shift, which can hinder the convergence and performance of deep learning models.

2.4 Sharpness Awareness Minimization

https://openreview.net/pdf?id=6Tm1mposlrM

3 Regularization

Ridge and LASSO regularization terms.

3.1 Early Stopping

3.2 Dropout

4 Neural Additive Models

Generalized additive models.

https://r.qcbs.ca/workshop08/book-en/gam-with-interaction-terms.html

https://arxiv.org/pdf/2004.13912.pdf

5 Lipshitz Regularity of Deep Neural Networks

Deep neural networks are known for being overparameterized and tends to predict data very nicely, known as benign overfitting. In fact, it can be proved that a data set of any size, we can always fit a one-layer perceptron that perfectly fits through all of them, given that the layer is large enough. In most cases, we are interested in fitting the data *smoothly* in the sense that data extrapolations are stable, i.e. a small perturbation of x should result in a small perturbation of $h(x)$. It turns out that the more parameters it has, the better this stability is and therefore the more robust the model.

Deep neural networks, despite their usefulness in many problems, are known for being very sensitive to their input. Adversarial examples take advantage of this weakness by adding carefully chosen perturbations that drastically change the output of the network. Adversarial machine learning attempts to study these weaknesses and hopefully use them to create more robust models. It is natural to expect that the precise configuration of the minimal necessary perturbations is a random artifact of the normal variability that arises in different runs of backpropagation learning. Yet, it has been found that adversarial examples are relatively robust, and are shared by neural networks with varied number of layers, activations or trained on different subsets of the training data. This suggest that the deep neural networks that are learned by backpropagation have *intrinsic* blind spots, whose structure is connected to the data distribution in a non-obvious way.

A metric to assess the robustness of a deep neural net $h_\theta : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ is its Lipshitz constant, which effectively bounds how much h can change given some change in $\mathbf{x}$.

Definition 5.1 (Lipshitz Continuity)**.** A function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ is called **Lipshitz continuous** if there exists a constant L such that for all $x, y \in \mathbb{R}^n$

$$||f(x) - f(y)||_2 \leq L||x - y||_2$$

and the smallest L for which the inequality is true is called the **Lipshitz constant**, denoted $\text{Lip}(f)$.

Theorem 5.1. If $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ is Lipschitz continuous, then

$$\text{Lip}(f) = \sup_{x \in \mathbb{R}^n} ||D_x f||_{\text{op}}$$

where $||\cdot||_{\text{op}}$ is the operator norm of a matrix. In particular, if f is scalar-valued, then its Lipschitz constant is the maximum norm of its gradient on its domain

$$\text{Lip}(f) = \sup_{x \in \mathbb{R}^n} ||\nabla f(x)||_2$$

The above theorem makes sense, since indeed the stability of the function should be equal to the stability of its "maximum" linear approximation $D_x f$.

Theorem 5.2 (Lipschitz Upper Bound for MLPs)**.** It has already been shown that for a K-layer MLP

$$h_\theta(\mathbf{x}) \coloneqq \mathbf{T}_K \circ \boldsymbol{\rho}_{K-1} \circ \mathbf{T}_{K-1} \circ \cdots \circ \boldsymbol{\rho}_1 \circ \mathbf{T}_1(\mathbf{x})$$

the Lipshitz constant for an affine map $\mathbf{T}_k(\mathbf{x}) = M_k \mathbf{x} + b_k$ is simply the operator norm (largest singular value) of M_k, while that of an activation function is always bounded by some well-known constant, usually 1. So, the Lipshitz constant of the entire composition h is simply the product of all operator norms of M_k.

What about K-computable functions in general? That is, given a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ with

$$\begin{aligned}
v_0(\mathbf{x}) &= \mathbf{x} \\
v_1(\mathbf{x}) &= g_1\big(v_0(\mathbf{x})\big) \\
v_2(\mathbf{x}) &= g_2\big(v_0(\mathbf{x}), v_1(\mathbf{x})\big) \\
\ldots &= \ldots \\
v_k(\mathbf{x}) &= g_k\big(v_0(\mathbf{x}), v_1(\mathbf{x}), \ldots, v_{k-1}(\mathbf{x})\big) \\
\ldots &= \ldots \\
v_K(\mathbf{x}) &= g_K\big(v_0(\mathbf{x}), v_1(\mathbf{x}), \ldots, v_{K-2}(\mathbf{x}), v_{K-1}(\mathbf{x})\big)
\end{aligned}$$

where $v_k : \mathbb{R}^n \longrightarrow \mathbb{R}^{n_k}$, with $n_0 = n$ and $n_K = m$, and

$$g_k : \prod_{i=0}^{k-1} \mathbb{R}^{n_i} \longrightarrow \mathbb{R}^{n_k}$$

To differentiate v_k w.r.t. $\mathbf{x}$, we can use the chain rule, resulting in the total derivative

$$\underbrace{\frac{\partial v_k}{\partial \mathbf{x}}}_{n_k \times n} = \sum_{i=1}^{k-1} \underbrace{\frac{\partial g_k}{\partial v_i}}_{n_k \times n_i} \underbrace{\frac{\partial v_i}{\partial \mathbf{x}}}_{n_i \times n}$$

Now we can compute the maximum iteratively.

1. First,
$$\frac{\partial v_0}{\partial \mathbf{x}} = I_{n \times n} \implies \bigg|\bigg|\frac{\partial v_0}{\partial \mathbf{x}}\bigg|\bigg|_2 = 1$$

2. Second,
$$\bigg|\bigg| \underbrace{\frac{\partial v_1}{\partial \mathbf{x}}}_{n_1 \times n} \bigg|\bigg| \leq \bigg|\bigg| \underbrace{\frac{\partial g_1}{\partial v_0}}_{n_1 \times n} \bigg|\bigg| \bigg|\bigg| \underbrace{\frac{\partial v_0}{\partial \mathbf{x}}}_{n \times n} \bigg|\bigg| = \bigg|\bigg|\frac{\partial g_1}{\partial v_0}\bigg|\bigg| \cdot ||I|| = \bigg|\bigg|\frac{\partial g_1}{\partial v_0}\bigg|\bigg|$$

3. Third,
$$\begin{aligned}
\bigg|\bigg| \underbrace{\frac{\partial v_2}{\partial \mathbf{x}}}_{n_2 \times n} \bigg|\bigg| &\leq \bigg|\bigg| \underbrace{\frac{\partial g_2}{\partial v_1}}_{n_2 \times n_1} \bigg|\bigg| \bigg|\bigg| \underbrace{\frac{\partial v_1}{\partial \mathbf{x}}}_{n_1 \times n} \bigg|\bigg| + \bigg|\bigg| \underbrace{\frac{\partial g_2}{\partial v_0}}_{n_2 \times n_0} \bigg|\bigg| \bigg|\bigg| \underbrace{\frac{\partial v_0}{\partial \mathbf{x}}}_{n_0 \times n} \bigg|\bigg| \\
&= \bigg|\bigg|\frac{\partial g_2}{\partial v_1}\bigg|\bigg| \bigg|\bigg|\frac{\partial g_1}{\partial v_0}\bigg|\bigg| + \bigg|\bigg|\frac{\partial g_2}{\partial v_0}\bigg|\bigg| 1
\end{aligned}$$

4. and so on, where we have calculated all previous $\frac{\partial v_i}{\partial \mathbf{x}}$ for $i \in [k-1]$, and we just need to compute $\frac{\partial g_k}{\partial v_i}$ for $i \in [k-1]$.

6 Convolutional Neural Networks

7 Radial Basis Function Neural Nets

8 Recurrent Neural Nets