# Neural Networks

Matt R

June 24, 2022

## Contents

# Part I

# Neural Networks and Deep Learning

# 1 Logistic Regression

We begin with a review of binary classification and logistic regression. To this end, suppose we have we have training examples $x \in \mathbb{R}^{m\times n}$ with binary labels $y \in \{0,1\}^{1\times n}$. We desire to train a model which yields an output $a$ which represents

$$a = \mathbb{P}(y = 1|x).$$

To this end, let $\sigma : \mathbb{R} \to (0,1)$ denote the sigmoid function, i.e.,

$$\sigma(z) = \frac{1}{1+e^{-z}},$$

and let $w \in \mathbb{R}^m$, $b \in \mathbb{R}$, and let

$$a = \sigma(w^T x + b).$$

To analyze the accuracy of model, we need a way to compare $y$ and $a$, and ideally this functional comparison can be optimized with respect to $(w,b)$ in such a way to minimize the error. To this end, we note that

$$\mathbb{P}(y|x) = a^y(1-a)^{1-y},$$

or rather

$$\mathbb{P}(y = 1|x) = a, \qquad \mathbb{P}(y = 0|x) = 1-a,$$

so $\mathbb{P}(y|x)$ represents the corrected probability. Now since we want

$$a \approx 1 \quad \text{when } y = 1,$$

and

$$a \approx 0 \quad \text{when } y = 0,$$

and $0 \leq a \leq 1$, any error using differences won't be refined enough to analyze when tuning the model. Moreover, since introducing the sigmoid function, our usual mean-squared-error function won't be convex. This leads us to apply the log function, which when restricted to $(0,1)$ is a bijective mapping of $(0,1) \to (-\infty, 0)$. This leads us to define our log-loss function

$$\begin{aligned}
\mathbb{L}(a,y) &= -\log(\mathbb{P}(y|x)) \\
&= -\log\left(a^y(1-a)^{1-y}\right) \\
&= -\left[y\log(a) + (1-y)\log(1-a)\right],
\end{aligned}$$

and finally, since we wish to analyze how our model performs on the entire training set, we need to average our log-loss functions to obtain our cost function $\mathbb{J}$ defined by

$$\begin{aligned}
\mathbb{J}(w,b) &= \frac{1}{n}\sum_{j=1}^{n}\mathbb{L}(a_j,y_j) \\
&= -\frac{1}{n}\sum_{j=1}^{n}\left[y_j\log(a_j)+(1-y_j)\log(1-a_j)\right] \\
&= -\frac{1}{n}\sum_{j=1}^{n}\left[y_j\log(\sigma(w^Tx_j+b))+(1-y_j)\log(1-\sigma(w^Tx_j+b))\right].
\end{aligned}$$

## 1.1 The Gradient

To compute the gradient of our cost function $\mathbb{J}$, we first write $\mathbb{J}$ as a sum of compositions as follows: We have the log-loss function considered as a map $\mathbb{L}:(0,1)\times\mathbb{R}\to\mathbb{R}$,

$$\mathbb{L}(a,y) = -\left[y\log(a)+(1-y)\log(1-a)\right],$$

we have the sigmoid function $\sigma:\mathbb{R}\to(0,1)$ with $\sigma(z)=a$ and $\sigma'(z)=a(1-a)$, and we have the collection of affine-functionals $\phi_x:\mathbb{R}^m\times\mathbb{R}\to\mathbb{R}$ given by

$$\phi_x(w,b)=w^Tx+b,$$

/Users/matt/Dropbox/notes/Neural Networks/neuralNetworks/src/python/utils.py for which we fix an arbitrary $x\in\mathbb{R}^m$ and write $\phi=\phi_x$, and set $z=\phi(w,b)$. Finally, we introduce the auxiliary function $\mathcal{L}:\mathbb{R}^m\times\mathbb{R}\to\mathbb{R}$ given by

$$\mathcal{L}(w,b)=\mathbb{L}(\sigma(\phi(w,b)),y).$$

Then by the chain rule, we have that

$$\begin{aligned}
d\mathcal{L} &= d_a\mathbb{L}(a,y)\circ d\sigma(z)\circ d_w\phi(w,b) \\
&= \left[-\frac{y}{a}+\frac{1-y}{1-a}\right]\cdot a(1-a)\cdot\begin{bmatrix}x^T & 1\end{bmatrix} \\
&= \left[-y(1-a)+a(1-y)\right]\cdot\begin{bmatrix}x^T & 1\end{bmatrix} \\
&= (a-y)\begin{bmatrix}x^T & 1\end{bmatrix}
\end{aligned}$$

Composition turns into matrix multiplication in the tangent space.

Moreover, for function $f : \mathbb{R}^N \to \mathbb{R}$ in Euclidean space, we have that $\nabla f = (df)^T$, and hence that

$$\nabla \mathcal{L}(w, b) = (a - y) \begin{bmatrix} x \\ 1 \end{bmatrix},$$

or rather

$$\partial_w \mathbb{L}(a, y) = (a - y)x, \qquad \partial_b \mathbb{L}(a, y) = a - y.$$

Finally, since our cost function $\mathbb{J}$ is the sum-log-loss, we have by linearity that

$$\begin{aligned} \partial_w \mathbb{J}(w, b) &= \frac{1}{n} \sum_{j=1}^{n} (a_j - y_j) x_j \\ &= \frac{1}{n} ((a - y) \cdot x^T)^T \\ &= \frac{1}{n} x \cdot (a - y)^T \end{aligned}$$

and

$$\partial_b \mathbb{J}(w, b) = \frac{1}{n} \sum_{j=1}^{n} (a_j - y_j).$$

## 1.2 Implementation in Python via numpy

Here we include the general method of coding a logistic regression model with $L^2$-regularization via the classical numpy library.

```
#! python3

import numpy as np

from mlLib.utils import apply_activation

class LinearParameters():
    def __init__(self, dims, bias=True, seed=1):
        """
        Parameters:
        -----------
        dims : tuple(int, int)
        bias : Boolean
            Default : True
        seed : int
            Default : 1

        Returns:
        --------
        None
        """
        np.random.seed(seed)
        self.dims = dims
        self.bias = bias
        self.w = np.random.randn(*dims) * 0.01
        if bias:
            self.b = np.zeros((dims[0], 1))

    def forward(self, x):
        """
        Parameters:
        -----------
        x : array_like

        Returns:
        --------
        z : array_like
        """
        z = np.einsum(’ij,jk’, self.w, x)
        if self.bias:
            z += self.b

        return z

    def backward(self, dz, x):
        """
        Parameters:
        -----------
        dz : array_like
        x : array_like

        Returns:
        --------
        None
        """
        if self.bias:
            self.db = np.sum(dz, axis=1, keepdims=True)
            assert (self.db.shape == self.b.shape)

        self.dw = np.einsum(’ij,kj’, dz, x)
        assert (self.dw.shape == self.w.shape)

    def update(self, learning_rate=0.01):
        """
        Parameters:
        -----------
        learning_rate : float
            Default : 0.01

        Returns:
        --------
        None
        """
        w = self.w - learning_rate * self.dw
        self.w = w

        if self.bias:
            b = self.b - learning_rate * self.db
            self.b = b

class LogisticRegression():
    def __init__(self, lp_reg):
        """
        Parameters:
        lp_reg : int
            2 : L_2 Regularization is imposed
            1 : L_1 Regularization is imposed
            0 : No regulariation is imposed

        Returns:
        --------
        None
        """
        self.lp_reg = lp_reg

    def predict(self, params, x):
        """
        Parameters:
        -----------
        params : class[LinearParameters]
        x : array_like

        Returns:
        --------
        a : array_like
        dg : array_like
        """
        z = params.forward(x)
        a, dg = apply_activation(z, 'sigmoid')
        return a, dg

    def cost_function(self, params, x, y, lambda_=0.01, eps=1e-8):
        """
        Parameters:
        -----------
        params : class[LinearParameters]
        x : array_like
        y : array_like
        lambda_ : float
            Default : 0.01
        eps : float
            Default : 1e-8

        Returns:
        --------
        cost : float
        """
        n = y.shape[1]

        R = np.sum(np.abs(params.w) ** self.lp_reg)
        R *= (lambda_ / (2 * n))

        a, _ = self.predict(params, x)
        a = np.clip(a, eps, 1 - eps)

        J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))

        cost = float(np.squeeze(J + R))

        return cost

    def fit(self, x, y, learning_rate=0.1, lambda_=0.01, seed=1, num_iters=10000):
        """
        Parameters:
        -----------
        x : array_like
        y : array_like
        learning_rate : float
            Default : 0.1
        lambda_ : float
            Default : 0.0
        num_iters : int
            Default : 10000

        Returns:
        --------
        costs : List[floats]
        params : class[Parameters]
        """
        dims = (y.shape[0], x.shape[0])
        n = x.shape[1]
        params = LinearParameters(dims, True, seed)

        if self.lp_reg == 0:
            lambda_ = 0.0

        costs = []
        for i in range(num_iters):
            a, _ = self.predict(params, x)
            cost = self.cost_function(params, x, y, lambda_)
            costs.append(cost)
            dz = (a - y) / n
            params.backward(dz, x)
            params.update(learning_rate)

            if i % 1000 == 0:
                print(f'Cost␣after␣iteration␣{i}:␣{cost}')

        return params

    def evaluate(self, params, x):
        """
        Parameters:
        -----------
        params : class[Parameters]
        x : array_like

        Returns:
        --------
        y_hat : array_like
        """
        a, _ = self.predict(params, x)
        y_hat = (~(a < 0.5)).astype(int)

        return y_hat

    def accuracy(self, params, x, y):
        """
        Parameters:
        -----------
        params : class[Parameters]
        x : array_like
        y : array_like

        Returns:
        --------
        accuracy : float
        """
        y_hat = self.evaluate(params, x)

        accuracy = np.sum(y_hat == y) / y.shape[1]

        return accuracy
```

## 1.3 Implementation in Python via sklearn

Here we include the general method of coding a logistic regression model via scikit-learn’s modeling library.

```
#! python3

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

def main(csv):
    df = pd.read_csv(csv)
    dataset = df.values
    x = dataset[:, :10]
    y = dataset[:, 10]

    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
    mu = np.mean(x, axis=0, keepdims=True)
    var = np.var(x, axis=0, keepdims=True)
    x_train = (x_train - mu) / np.sqrt(var)
    x_test = (x_test - mu) / np.sqrt(var)

    log_reg = LogisticRegression()
    log_reg.fit(x_train, y_train)
    train_acc = log_reg.score(x_train, y_train)
    print(f’The␣accuracy␣on␣the␣training␣set:␣{train_acc}.’)
    test_acc = log_reg.score(x_test, y_test)
    print(f’The␣accuracy␣on␣the␣test␣set:␣{test_acc}.’)
```

## 2 Neural Networks: A Single Hidden Layer

Suppose we wish to consider the binary classification problem given the training set $(x, y)$ with $x \in \mathbb{R}^{m_0 \times n}$ and $y \in \{0,1\}^{1\times n}$. Usually with logistic regression we have the following type of structure:

$$[x^1, ..., x^{m_0}] \xrightarrow{\varphi} [z] \xrightarrow{g} [a] \xrightarrow{=} \hat{y},$$

where

$$z = \varphi(x) = w^T x + b,$$

is our affine-linear transformation, and

$$a = g(z) = \sigma(z)$$

is our sigmoid function. Such a structure will be called a *network*, and the $[a]$ is known as the *activation node*. Logistic regression can be too simplistic of a model for many situations, e.g., if the dataset isn't linearly separable (i.e., there doesn't exist some well-defined decision boundary built from a linear-surface), then logistic regression won't give a high-accuracy model. To modify this model to handle more complex situations, we introduce a new "hidden layer" of nodes with their own (possibly different) activation functions. That is, we consider a network of the following form:

$$\underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{m_0}\end{bmatrix}}_{\text{Layer 0}} \xrightarrow{\varphi^{[1]}} \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]m_1}\end{bmatrix} \xrightarrow{g^{[1]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]m_1}\end{bmatrix}}_{\text{Layer 1}} \xrightarrow{\varphi^{[2]}} \underbrace{\begin{bmatrix} z^{[2]}\end{bmatrix} \xrightarrow{g^{[2]}} \begin{bmatrix} a^{[2]}\end{bmatrix}}_{\text{Layer 2}} \xrightarrow{=} \hat{y},$$

where

$$\varphi^{[1]} : \mathbb{R}^{m_0} \to \mathbb{R}^{m_1}, \qquad \varphi^{[1]}(x) = W^{[1]}x + b^{[1]},$$
$$\varphi^{[2]} : \mathbb{R}^{m_1} \to \mathbb{R}, \qquad \varphi^{[2]}(x) = W^{[2]}x + b^{[2]},$$

and $W^{[1]} \in \mathbb{R}^{m_1 \times m_0}, W^{[2]} \in \mathbb{R}^{1\times m_1}, b^{[1]} \in \mathbb{R}^{m_1}, b^{[2]} \in \mathbb{R}$, and $g^{[\ell]}$ is a *broadcasted* activator function (e.g., the sigmoid function $\sigma(z)$, or $\tanh(z)$, or $\mathrm{ReLU}(z)$). Such a network is called a 2-layer neural network where $x$ is the input layer (called layer-0), $a^{[1]}$ is a hidden layer (called layer-1), and $a^{[2]}$ is the output layer (called layer-2).

**Definition 2.1.** *Suppose $g : \mathbb{R} \to \mathbb{R}$ is any function. Then we say $G : \mathbb{R}^m \to \mathbb{R}^m$ is the **broadcast** of $g$ from $\mathbb{R}$ to $\mathbb{R}^m$ if*

$$\begin{aligned} G(v) &= G(v^i e_i) \\ &= g(v^i)e_i, \end{aligned}$$

*where $v \in \mathbb{R}^m$ and $\{e_i : 1 \leq i \leq m\}$ is the standard basis for $\mathbb{R}^m$. In practice, we will write $g = G$ for a broadcasted function, and let the context determine the meaning of $g$.*

castingDifferential

**Lemma 2.2.** *Suppose $g : \mathbb{R} \to \mathbb{R}$ is any smooth function and $G : \mathbb{R}^m \to \mathbb{R}^m$ is the broadcasting of $g$ from $\mathbb{R}$ to $\mathbb{R}^m$. Then the differential $dG_z : T_z\mathbb{R}^m \to T_{G(z)}\mathbb{R}^m$ is given by*

$$dG_z(v) = [g'(z^i)] \odot [v^i],$$

*where $\odot$ is the Hadamard product (also know as component-wise multiplication), and has matrix-representation in $\mathbb{R}^{m\times m}$ given by*

$$[dG_z]^i_j = \delta^i_j g'(z^i).$$

**Proof:** We calculate

$$\begin{aligned} dG_z(v) &= \frac{d}{dt}\bigg|_{t=0} G(z + tv) \\ &= \frac{d}{dt}\bigg|_{t=0} (g(z^i + tv^i)) \\ &= (g'(z^i)v^i) \\ &= [g'(z^i)] \odot [v^i], \end{aligned}$$

and letting $e_1, ...e_m$ denote the usual basis for $T_z\mathbb{R}^m$ (identified with $\mathbb{R}^m$), we see that

$$\begin{aligned} dG_z(e_j) &= [g'(z^i)] \odot e_j \\ &= g'(z^j)e_j, \end{aligned}$$

from which conclude that $dG_z$ is diagonal with $(j, j)$-th entry $g'(z^j)$ as desired. □

Returning to our network, let us lay out all of these functions explicitly (in the Smooth Category) as to facilitate our later computations for our cost function and our gradients. To this end:

$$\begin{aligned} &\varphi^{[1]} : \mathbb{R}^{m_0} \to \mathbb{R}^{m_1}, & &d\varphi^{[1]} : T\mathbb{R}^{m_0} \to T\mathbb{R}^{m_1}, \\ &z^{[1]} = \varphi^{[1]}(x) = W^{[1]}x + b^{[1]}, & &d\varphi^{[1]}_x(v) = W^{[1]}v; \end{aligned}$$

$$g^{[1]} : \mathbb{R}^{m_1} \to \mathbb{R}^{m_1}, \qquad dg^{[1]} : T\mathbb{R}^{m_1} \to T\mathbb{R}^{m_1},$$

$$a^{[1]} = g^{[1]}(z^{[1]}), \qquad \frac{\partial a^{[1]\mu}}{\partial z^{[1]\nu}} = \delta^{\mu}_{\nu} g^{[1]\prime}(z^{[1]\mu});$$

$$\varphi^{[2]} : \mathbb{R}^{m_1} \to \mathbb{R}^{m_2}, \qquad d\varphi^{[2]} : T\mathbb{R}^{m_1} \to T\mathbb{R}^{m_2},$$

$$z^{[2]} = \varphi^{[2]}(a^{[1]}) = W^{[2]}a^{[1]} + b^{[2]}, \qquad d\varphi^{[2]}_{a^{[2]}}(v) = W^{[2]}v;$$

$$g^{[2]} : \mathbb{R}^{m_2} \to \mathbb{R}^{m_2}, \qquad dg^{[2]} : T\mathbb{R}^{m_2} \to T\mathbb{R}^{m_2},$$

$$a^{[2]} = g^{[2]}(z^{[2]}), \qquad \frac{\partial a^{[2]\mu}}{\partial z^{[2]\nu}} = \delta^{\mu}_{\nu} g^{[2]\prime}(z^{[2]\mu}).$$

That is, given an input $x \in \mathbb{R}^{m_0}$, we get a predicted value $\hat{y} \in \mathbb{R}^{m_2}$ of the form

$$\hat{y} = g^{[2]} \circ \varphi^{[2]} \circ g^{[1]} \circ \varphi^{[1]}(x).$$

This compositional function is known as *forward propagation.*

## 2.1 Backward Propagation

backPropDerivation

Since we wish to optimize our model with respect to our parameter $W^{[\ell]}$ and $b^{[\ell]}$, we consider a generic loss function $\mathbb{L} : \mathbb{R}^{m_2} \times \mathbb{R}^{m_2} \to \mathbb{R}$, $\mathbb{L}(\hat{y}, y)$, and by acknowledging the potential abuse of notation, we assume $y$ is fixed, and consider the aforementioned as a function of a single-variable

$$\mathbb{L}_y : \mathbb{R}^{m_2} \to \mathbb{R}, \qquad \mathbb{L}_y(\hat{y}) = \mathbb{L}(\hat{y}, y).$$

We also define the function

$$\Phi(A, u, \xi) = A\xi + u,$$

and note that we're suppressing a dependence on the layer $\ell$ which only affects our domain and range of $\Phi$ (and not the actual calculations involving the derivatives). Moreover, in coordinates we see that

$$\begin{aligned} \frac{\partial \Phi^i}{\partial A^{\mu}_{\nu}} &= \frac{\partial}{\partial A^{\mu}_{\nu}}(A^i_j \xi^j + u^i) \\ &= (\delta^i_{\mu} \delta^{\nu}_j \xi^j) \\ &= \delta^i_{\mu} \xi^{\nu}; \end{aligned}$$

$$\begin{aligned}\frac{\partial \Phi^i}{\partial u^\mu} &= \frac{\partial}{\partial u^\mu}(A^i_j \xi^j + u^i) \\ &= \delta^i_\mu;\end{aligned}$$

and

$$\begin{aligned}\frac{\partial \Phi^i}{\xi^\mu} &= \frac{\partial}{\partial \xi^\mu}(A^i_j \xi^j + u^i) \\ &= A^i_j \delta^j_\mu \\ &= A^i_\mu.\end{aligned}$$

We now define the compositional function

$$F : \mathbb{R}^{m_2 \times m_1} \times \mathbb{R}^{m_2} \times \mathbb{R}^{m_1 \times m_0} \times \mathbb{R}^{m_1} \times \mathbb{R}^{m_0} \to \mathbb{R}$$

given by

$$F(C, c, B, b, x) = \mathbb{L}_y \circ g^{[2]} \circ \Phi \circ (\mathbb{1}_{\mathbb{R}^{m_2 \times m_1}} \times \mathbb{1}_{\mathbb{R}^{m_2}} \times (g^{[1]} \circ \Phi))(C, c, B, b, x).$$

We first introduce an error term $\delta^{[2]} \in \mathbb{R}^{m_2}$ defined by

$$\begin{aligned}\delta^{[2]} :&= \nabla(\mathbb{L}_y \circ g^{[2]})(z^{[2]}) \\ &= (d\mathbb{L}_y \circ g^{[2]})_{z^{[2]}})^T.\end{aligned}$$

Now we calculate the gradient $\frac{\partial F}{\partial C}$ in coordinates by $\delta^{[2]} = d_{z^{[2]}}F$

$$\begin{aligned}\frac{\partial F}{\partial C^\mu_\nu} &= \frac{\partial}{\partial C^\mu_\nu}\left[\mathbb{L}_y \circ g^{[2]} \circ \Phi(C, c, a^{[1]})\right] \\ &= \sum_{j=1}^{m_2} \delta^{[2]j} \frac{\partial}{\partial C^\mu_\nu}(C^j_i a^{[1]i} + c^j) \\ &= \sum_{j=1}^{m_2} \delta^{[2]j} \delta^j_\mu a^{[1]\nu} \\ &= \delta^{[2]}{}_\mu a^{[1]\nu} \\ &= [a^{[1]} \delta^{[2]T}]^\nu_\mu\end{aligned}$$

and hence that

$$\begin{aligned}\frac{\partial F}{\partial C} &= \left[\frac{\partial F}{\partial C^\mu_\nu}\right]^T \\ &= \left[\delta^{[2]}_\mu a^{[1]\nu}\right]^T \\ &= \delta^{[2]} a^{[1]T}.\end{aligned}$$

Moreover, we also calculate

$$\frac{\partial F}{\partial c^\mu} = \sum_{j=1}^{m_2} \delta^{[2]j} \delta^j_\mu,$$

and hence that

$$\frac{\partial F}{\partial c} = \delta^{[2]}.$$

Next we introduce another error term $\delta^{[1]} \in \mathbb{R}^{m_1}$ defined by

$$\delta^{[1]} = [dg^{[1]}_{z^{[1]}}]^T C^T \delta^{[2]}$$

with coordinates $\delta^{[1]} = d_{z^{[1]}} F$

$$\begin{aligned}
(\delta^{[1]\mu})^T &= \sum_{i=1}^{m_2} \sum_{j=1}^{m_1} \delta^{[2]i} C^i_j g^{[1]\prime}(z^{[1]j}) \delta^j_\mu \\
&= \sum_{i=1}^{m_2} \delta^{[2]i} C^i_\mu g^{[1]\prime}(z^{[1]\mu})
\end{aligned}$$

and now calculate the gradient $\frac{\partial F}{\partial B}$ in coordinates by

$$\begin{aligned}
\frac{\partial F}{\partial B^\mu_\nu} &= \frac{\partial}{B^\mu_\nu} \left[ \mathbb{L}_y \circ g^{[2]} \circ \Phi(C, c, g^{[1]}(Bx + b)) \right] \\
&= \sum_{j=1}^{m_2} \delta^{[2]j} \sum_{\rho=1}^{m_1} \frac{\partial \Phi^j}{\partial \xi^\rho} \sum_{\lambda=1}^{m_1} \frac{\partial a^{[1]\rho}}{\partial z^{[1]\lambda}} \frac{\partial \Phi^\lambda}{\partial B^\mu_\nu} \\
&= \sum_{j=1}^{m_2} \delta^{[2]j} \sum_{\rho=1}^{m_1} \frac{\partial \Phi^j}{\partial \xi^\rho} \sum_{\lambda=1}^{m_1} \delta^\rho_\lambda g^{[1]\prime}(z^{[1]\rho}) \delta^\lambda_\mu x^\nu \\
&= \sum_{j=1}^{m_2} \delta^{[2]j} \sum_{\rho=1}^{m_1} \frac{\partial \Phi^j}{\partial \xi^\rho} \delta^\rho_\mu g^{[1]\prime}(z^{[1]\rho}) x^\nu \\
&= \sum_{j=1}^{m_2} \delta^{[2]j} \sum_{\rho=1}^{m_1} C^j_\rho \delta^\rho_\mu g^{[1]\prime}(z^{[1]\rho}) x^\nu \\
&= \sum_{j=1}^{m_2} \delta^{[2]j} C^j_\mu g^{[1]\prime}(z^{[1]\mu}) x^\nu \\
&= \delta^{[1]}{}_\mu x^\nu \\
&= \left[ x \delta^{[1]T} \right]^\nu_\mu,
\end{aligned}$$

and hence that

$$\begin{aligned}\frac{\partial F}{\partial B} &= \left[\frac{\partial F}{\partial B^{\mu}_{\nu}}\right]^T \\ &= \delta^{[2]} x^T.\end{aligned}$$

Moreover, from the above calculation, we immediately see that

$$\frac{\partial F}{\partial b^{\mu}} = \delta^{[1]}.$$

In summary, we've computed the following gradients

$$\begin{aligned}\frac{\partial F}{\partial W^{[2]}} &= \delta^{[2]} a^{[1]T} \\ \frac{\partial F}{\partial b^{[2]}} &= \delta^{[2]} \\ \frac{\partial F}{\partial W^{[1]}} &= \delta^{[1]} x^T \\ \frac{\partial F}{\partial b^{[1]}} &= \delta^{[1]},\end{aligned}$$

where

$$\begin{aligned}\delta^{[2]} &= [d(\mathbb{L}_y \circ g^{[2]})_{z^{[2]}}]^T \\ \delta^{[1]} &= [dg^{[1]}_{z^{[1]}}]^T C^T \delta^{[2]}.\end{aligned}$$

Finally, we recall that our cost function $\mathbb{J}$ is the average sum of our loss function $\mathbb{L}$ over our training set, we get that

$$\mathbb{J}(W^{[2]}, b^{[2]}, W^{[1]}, b^{[1]}) = \frac{1}{n} \sum_{j=1}^{n} F(W^{[2]}, b^{[2]}, W^{[1]}, b^{[1]}, x_j),$$

and hence that

$$\begin{aligned}\frac{\partial \mathbb{J}}{\partial W^{[2]}} &= \frac{1}{n} \sum_{j=1}^{n} \delta^{[2]}{}_j a^{[1]}{}_j{}^T = \frac{1}{n} \delta^{[2]} a^{[1]T} \\ \frac{\partial \mathbb{J}}{\partial b^{[2]}} &= \frac{1}{n} \sum_{j=1}^{n} \delta^{[2]}{}_j \\ \frac{\partial \mathbb{J}}{\partial W^{[1]}} &= \frac{1}{n} \sum_{j=1}^{n} \delta^{[1]}{}_j x_j^T = \frac{1}{n} \delta^{[1]} x^T \\ \frac{\partial \mathbb{J}}{\partial b^{[1]}} &= \frac{1}{n} \sum_{j=1}^{n} \delta^{[1]}{}_j\end{aligned}$$

## 2.2 Activation Functions

There are mainly only a handful of activating functions we consider for our non-linearity conditions.

### 2.2.1 The Sigmoid Function

We have the sigmoid function $\sigma(z)$ given by

$$\sigma : \mathbb{R} \to (0,1), \qquad \sigma(z) = \frac{1}{1+e^{-z}}.$$

We note that since

$$\begin{aligned} 1 - \sigma(z) &= 1 - \frac{1}{1+e^{-z}} \\ &= \frac{e^{-z}}{1+e^{-z}} \end{aligned}$$

$$\begin{aligned} \sigma'(z) &= \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} \\ &= \sigma(z)(1-\sigma(z)) \end{aligned}$$

Moreover, suppose that $g : \mathbb{R}^m \to \mathbb{R}^m$ is the broadcasting of $\sigma$ from $\mathbb{R}$ to $\mathbb{R}^m$, then for $z = (z^1, ..., z^m) \in \mathbb{R}^m$, we have that

$$g(z) = (\sigma(z^i)),$$

and $dg_z : T_z\mathbb{R}^m \to T_{g(z)}\mathbb{R}^m$ given by

$$\begin{aligned} dg_z(v) &= \left.\frac{d}{dt}\right|_{t=0} g(z+tv) \\ &= \left.\frac{d}{dt}\right|_{t=0} (\sigma(z^i + tv^i)) \\ &= (\sigma'(z^i)v^i) \\ &= (\sigma(z^i)(1-\sigma(z^i))v^i) \\ &= g(z) \odot (1 - g(z)) \odot v, \end{aligned}$$

where $\odot$ represents the Hadamard product (or component-wise multiplication); or rather, as as a matrix in $\mathbb{R}^{m \times m}$,

$$[dg_z]^\mu_\nu = \delta^\mu_\nu \sigma(z^\mu)(1-\sigma(z^\mu)).$$

### 2.2.2 The Hyperbolic Tangent Function

We have the hyperbolic tangent function $\tanh(z)$ given by

$$\tanh : \mathbb{R} \to (-1, 1), \qquad \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

We then calculate

$$\begin{aligned}\tanh'(z) &= \frac{(e^z + e^{-z})(e^z + e^{-z}) - (e^z - e^{-z})(e^z - e^{-z})}{(e^z + e^{-z})^2} \\ &= \frac{(e^z + e^{-z})^2}{(e^z + e^{-z})^2} - \frac{e^z - e^{-z})^2}{(e^z + e^{-z})^2} \\ &= 1 - \tanh^2(z).\end{aligned}$$

Suppose $g : \mathbb{R}^m \to \mathbb{R}^m$ is the broadcasting of tanh from $\mathbb{R}$ to $\mathbb{R}^m$, then for $z = (z^1, ..., z^m) \in \mathbb{R}^m$, we have that

$$g(z) = (\tanh(z^i)),$$

and $dg_z : T_z\mathbb{R}^m \to T_{g(z)}\mathbb{R}^m$ given by

$$\begin{aligned}dg_z(v) &= [\tanh'(z^i)] \odot [v^i] \\ &= [1 - \tanh^2(z^i)] \odot [v^i] \\ &= \delta^i_j(1 - \tanh^2(z^i))v^j.\end{aligned}$$

### 2.2.3 The Rectified Linear Unit Function

We have the leaky-ReLU function $\text{ReLU}(z; \beta)$ given by

$$\text{ReLU} : \mathbb{R} \to \mathbb{R}, \qquad \text{ReLU}(z; \beta) = \max\{\beta z, z\},$$

for some $\beta > 0$ (typically chosen very small).

We have the rectified linear unit function $\text{ReLU}(z)$ given by setting $\beta = 0$ in the leaky-ReLu function, i.e.,

$$\text{ReLU} : \mathbb{R} \to [0, \infty), \qquad \text{ReLU}(z) = \text{ReLU}(z; \beta = 0) = \max\{0, z\}.$$

We then calculate

$$\begin{aligned}\text{ReLU}'(z; \beta) &= \begin{cases}\beta & z < 0 \\ 1 & z \geq 0\end{cases} \\ &= \beta\chi_{(-\infty,0)}(z) + \chi_{[0,\infty)}(z),\end{aligned}$$

where

$$\chi_A(z) = \begin{cases} 1 & z \in A \\ 0 & z \notin A \end{cases},$$

is the indicator function.

Suppose $g : \mathbb{R}^m \to \mathbb{R}^m$ is the broadcasting of ReLU from $\mathbb{R}$ to $\mathbb{R}^m$. Then for $z = (z^1, ..., z^m) \in \mathbb{R}^m$, we have that

$$g(z) = \mathrm{ReLU}(z^i; \beta),$$

and $dg_z : T_z\mathbb{R}^m \to T_{g(z)}\mathbb{R}^m$ given by

$$\begin{aligned} dg_z(v) &= [\mathrm{ReLU}'(z^i; \beta)] \odot [v^i] \\ &= \delta^i_j(\beta\chi_{(-\infty,0)}(z^i) + \chi_{[0,\infty)}(z^i))v^j. \end{aligned}$$

### 2.2.4 The Softmax Function

We finally have the softmax function softmax$(z)$ given by

$$\mathrm{softmax} : \mathbb{R}^m \to \mathbb{R}^m, \qquad \mathrm{softmax}(z) = \frac{1}{\sum_{j=1}^m e^{z^j}} \begin{pmatrix} e^{z^1} \\ e^{z^2} \\ \vdots \\ e^{z^m} \end{pmatrix},$$

which we typically use on our outer-layer to obtain a probability distribution over our predicted labels. Let

$$S^i = x^i \circ \mathrm{softmax}(z),$$

denote the $i$-th component of softmax$(z)$, and so we calculate

$$\begin{aligned} \frac{\partial S^i}{\partial z^j} &= \frac{\partial}{\partial z^j}\left[\left(\sum_{k=1}^m e^{z^k}\right)^{-1} e^{z^i}\right] \\ &= -\left(\sum_{k=1}^m e^{z^k}\right)^{-2}\left(\sum_{k=1}^m e^{z^k}\delta^k_j\right)e^{z^i} + \left(\sum_{k=1}^m e^{z^k}\right)^{-1} e^{z^i}\delta^i_j \\ &= -\left(\sum_{k=1}^m e^{z^k}\right)^{-2} e^{z^j}e^{z^i} + S^i\delta^i_j \\ &= -S^jS^i + S^i\delta^i_j \\ &= S^i(\delta^i_j - S^j). \end{aligned}$$

That is, as a map $dS_z : T_z\mathbb{R}^m \to T_{S(z)}\mathbb{R}^m$, we have that

$$dS_z = [S^i(\delta^i_j - S_j)]^i_j,$$

and we make note that $dS_z$ is symmetric.

## 2.3 Binary Classification - An Example

We return the network given by

$$\underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{m_0} \end{bmatrix}}_{\text{Layer 0}} \xrightarrow{\varphi^{[1]}} \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]m_1} \end{bmatrix} \xrightarrow{g^{[1]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]m_1} \end{bmatrix}}_{\text{Layer 1}} \xrightarrow{\varphi^{[2]}} \underbrace{\begin{bmatrix} z^{[2]} \end{bmatrix} \xrightarrow{g^{[2]}} \begin{bmatrix} a^{[2]} \end{bmatrix}}_{\text{Layer 2}} \xrightarrow{=} \hat{y},$$

and show how such a model would be trained using python below. We assume layer-2 has the sigmoid function (since it's binary classification) as an activator and our hidden layer has the ReLU function as activators.

We note that $m_2 = 1$ since we're dealing with a single activator in this layer, and

$$a^{[2]} = g^{[2]}(z^{[2]}) = \sigma(z^{[2]}),$$

with

$$d(g^{[2]})_{z^{[2]}} = \sigma'(z^{[2]}) = \sigma(z^{[2]})(1 - \sigma(z^{[2]})) = a^{[2]}(1 - a^{[2]}).$$

In layer-1, we have that

$$a^{[1]} = g^{[1]}(z^{[1]}) = \text{ReLU}(z^{[1]}),$$

with

$$d(g^{[1]})_{z^{[1]}} = \left[\delta^\mu_\nu \chi_{[0,\infty)}(z^{[1]\mu})\right]^\mu_\nu .$$

Finally, we choose our loss function $\mathbb{L}(\hat{y}, y)$ to be the log-loss function (since we're using the sigmoid activator on the outer-layer), i.e.,

$$\mathbb{L}(\hat{y}, y) = -y\log(\hat{y}) - (1 - y)\log(1 - \hat{y}),$$

or rather

$$\mathbb{L}(x, y) = -y\log(a^{[2]}) - (1 - y)\log(1 - a^{[2]}).$$

We then have the cost function $\mathbb{J}$ given by

$$\begin{aligned}\mathbb{J}(W^{[2]}, b^{[2]}, W^{[1]}, b^{[1]}) &= \frac{-1}{n}\sum_{j=1}^{n}\big(y_j \log(a^{[2]}{}_j) + (1-y_j)\log(1-a^{[2]}{}_j)\big)\\ &= \frac{-1}{n}\big(\langle y, \log(a^{[2]})\rangle + \langle 1-y, \log(1-a^{[2]})\rangle\big)\end{aligned}$$

Moreover, when using backpropagation, we see that

$$\begin{aligned}\delta^{[2]T}_j &= d(\mathbb{L}_{y_j})_{a^{[2]}} \cdot d(g^{[2]})_{z^{[2]}{}_j}\\ &= \left(-\frac{y_j}{a^{[2]}{}_j} + \frac{1-y_j}{1-a^{[2]}{}_j}\right)\cdot (a^{[2]}{}_j(1-a^{[2]}{}_j)\\ &= a^{[2]}{}_j - y_j,\end{aligned}$$

or rather

$$\delta^{[2]} = a^{[2]} - y.$$

Similarly, we compute

$$\begin{aligned}\delta^{[1]T}_j &= \delta^{[2]T}_j W^{[2]}[dg^{[1]}_{z^{[1]}{}_j}]\\ &= \delta^{[2]T}_j W^{[2]}[\delta^{\mu}_{\nu} \cdot \chi_{[0,\infty)}(z^{[1]\mu}_j)]\end{aligned}$$

### 2.3.1 Random Initialization

In the section that follows, we see that to begin gradient descent for a shallow neural network, we initialize our parameters $b^{[\ell]}$ to be 0, but choose an arbitrarily small, but nonzero initialization for $W^{[\ell]}$. Let's see why we choose $W^{[\ell]}$ to be nonzero. Indeed, suppose we initialize with $b^{[\ell]} = 0$ and $W^{[\ell]} = 0$. Then we see that

$$\delta^{[1]T} = \delta^{[2]}W^{[2]}dg^{[1]}_{z^{[1]}} = 0,$$

and so

$$\frac{\partial \mathbb{J}}{\partial W^{[1]}} = \frac{1}{n}\delta^{[1]}x^T = 0.$$

Then we conclude that our parameter $W^{[1]}$ remains at 0 during every iteration which is enough reason to not initialize $W^{[2]}$ at 0. Similarly, since

$$a^{[1]} = \tanh(W^{[1]}x + b^{[1]}) = \tanh(0) = 0,$$

we reach a similar conclusion about $W^{[1]}$ and $W^{[2]}$, respectively.

# 3 Deep Neural Networks

In this section we discuss a general “deep” neural network, which consist of $L$ layers. That is, we have a network of the form:

$$\underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{m_0} \end{bmatrix}}_{\text{Layer 0}} \xrightarrow{\varphi^{[1]}} \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]m_1} \end{bmatrix} \xrightarrow{g^{[1]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]m_1} \end{bmatrix}}_{\text{Layer 1}} \xrightarrow{\varphi^{[2]}} \underbrace{\begin{bmatrix} z^{[2]1} \\ \vdots \\ z^{[2]m_2} \end{bmatrix} \xrightarrow{g^{[2]}} \begin{bmatrix} a^{[2]1} \\ \vdots \\ a^{[2]m_2} \end{bmatrix}}_{\text{Layer 2}} \xrightarrow{\varphi^{[3]}} \cdots$$

$$\cdots \xrightarrow{\varphi^{[L-1]}} \underbrace{\begin{bmatrix} z^{[L-1]1} \\ \vdots \\ z^{[L-1]m_{L-1}} \end{bmatrix} \xrightarrow{g^{[L-1]}} \begin{bmatrix} a^{[L-1]1} \\ \vdots \\ a^{[L-1]m_{L-1}} \end{bmatrix}}_{\text{Layer } L-1} \xrightarrow{\varphi^{[L]}} \underbrace{\begin{bmatrix} z^{[L]1} \\ \vdots \\ z^{[L]m_L} \end{bmatrix} \xrightarrow{g^{[L]}} \begin{bmatrix} a^{[L]1} \\ \vdots \\ a^{[L]m_L} \end{bmatrix}}_{\text{Layer } L} \xrightarrow{=} \begin{bmatrix} \hat{y}^1 \\ \vdots \\ \hat{y}^{m_L} \end{bmatrix},$$

where

$$m_\ell := \text{ the number of nodes in layer-}\ell,$$

$$\varphi^{[\ell]} : \mathbb{R}^{m_{\ell-1}} \to \mathbb{R}^{m_\ell}, \qquad \varphi^{[\ell]}(\xi) = W^{[\ell]}\xi + b^{[\ell]}, \qquad W^{[\ell]} \in \mathbb{R}^{m_\ell \times m_{\ell-1}}, b \in \mathbb{R}^{m_\ell},$$

and

$$g^{[\ell]} : \mathbb{R}^{m_\ell} \to \mathbb{R}^{m_\ell},$$

is a broadcasted activation function determined by the layer-$\ell$.

As with a shallow network, our functional composition to obtain $a^{[L]}$ is known as forward propagation.

## 3.1 Backward Propagation

As the general derivation for backpropagation can be easily (if not tediously) generalized from Section 2.1 using induction, we give the general outline for computational purposes.

Let $\mathbb{L} : \mathbb{R}^{m_L} \times \mathbb{R}^{m_L} \to \mathbb{R}$ be a generic loss function, and suppose our cost function is given by the usual

$$\mathbb{J}(W, b) = \frac{1}{n} \sum_{j=1}^{n} \mathbb{L}(\hat{y}_j, y_j).$$

Then from previous computations, we have the following gradients for any

$\ell \in \{1, 2, ..., L\}$, that

$$\frac{\partial \mathbb{J}}{\partial W^{[\ell]}} = \frac{1}{n}\delta^{[\ell]}a^{[\ell-1]T}$$
$$\frac{\partial \mathbb{J}}{\partial b^{[\ell]}} = \frac{1}{n}\sum_{j=1}^{n}\delta^{[\ell]}{}_j$$

where we impose the notation of

$$a^{[0]} := x.$$

So we need only give a full characterization of $\delta^{[\ell]}$.. To this end, we define recursively starting at layer-$L$ by

$$\begin{aligned}
\delta^{[L]T} &:= d(\mathbb{L}_y)_{a^{[L]}} \cdot dg^{[L]}_{z^{[L]}},\\
\delta^{[L-1]T} &:= \delta^{[L]T} \cdot W^{[L]} \cdot dg^{[L-1]}_{z^{[L-1]}},\\
&\vdots\\
\delta^{[\ell]T} &:= \delta^{[\ell+1]T}W^{[\ell+1]}dg^{[\ell]}_{z^{[\ell]}},\\
&\vdots\\
\delta^{[1]T} &:= \delta^{[2]T}W^{[2]}dg^{[1]}_{z^{[1]}},
\end{aligned}$$

as desired.

## 3.2 Implementation in Python via `numpy`

We implement a neural network with an arbitrary number of layers and nodes, with the ReLU function as the activator on all hidden nodes and the sigmoid function on the output layer for binary classification with the log-loss function.

```python
#! python3

import numpy as np

from mlLib.utils import LinearParameters, apply_activation

class NeuralNetwork():
    def __init__(self, config):
        """
        Parameters:
```

```
        -----------
        config : Dict
            config[’lp_reg’] = 0,1,2
            config[’nodes’] = List[int]
            config[’bias’] = List[Boolean]
            config[’activators’] = List[str]

        Returns:
        --------
        None
        """
        self.config = config
        self.lp_reg = config[’lp_reg’]
        self.nodes = config[’nodes’]
        self.bias = config[’bias’]
        self.activators = config[’activators’]
        self.L = len(config[’nodes’]) - 1

    def forward_propagation(self, params, x):
        """
        Parameters:
        -----------
        params : Dict[class[Parameters]]
            params[l].w = Weights
            params[l].bias = Boolean
            params[l].b = Bias
        x : array_like

        Returns:
        --------
        cache = Dict[array_like]
            cache[’a’] = a
            cache[’dg’] = dg

        """
        # Initialize dictionaries
        a = {}
        dg = {}

        a[0], dg[0] = apply_activation(x, self.activators[0])

        for l in range(1, self.L + 1):
            z = params[l].forward(a[l - 1])
            a[l], dg[l] = apply_activation(z, self.activators[l])

        cache = {’a’ : a, ’dg’ : dg}
        return cache

    def cost_function(self, params, a, y, lambda_=0.01, eps=1e-8):
        """
        Parameters:
        -----------
        params: class[Parameters]
        a: array_like
        y: array_like
        lambda_: float
            Default: 0.01
        eps: float
            Default: 1e-8

        Returns:
        --------
        cost: float
        """
        n = y.shape[1]
        if self.lp_reg == 0:
            lambda_ = 0.0

        # Compute regularization term
        R = 0
        for param in params.values():
            R += np.sum(np.abs(param.w) ** self.lp_reg)
        R *= (lambda_ / (2 * n))

        # Compute unregularized cost
        a = np.clip(a, eps, 1 - eps)    # Bound a for stability
        J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))

        cost = float(np.squeeze(J + R))

        return cost

    def backward_propagation(self, params, cache, y):
        """
        Parameters:
        -----------
        params : Dict[class[Parameters]]
            params[l].w = Weights
            params[l].bias = Boolean
            params[l].b = Bias
        cache : Dict[array_like]
            cache[’a’] : array_like
            cache[’dg’] : array_like
        y : array_like

        Returns:
        --------
        None
        """

        # Retrieve cache
        a = cache['a']
        dg = cache['dg']

        # Initialize differentials along the network
        delta = {}
        delta[self.L] = (a[self.L] - y) / y.shape[1]

        for l in reversed(range(1, self.L + 1)):
            delta[l - 1] = dg[l- 1] * params[l].backward(delta[l], a[l - 1])

    def update_parameters(self, params, learning_rate=0.1):
        """
        Parameters:
        -----------
        params : Dict[class[Parameters]]
            params[l].w = Weights
            params[l].bias = Boolean
            params[l].b = Bias
        learning_rate : float
            Default : 0.01

        Returns:
        --------
        None
        """
        for param in params.values():
            param.update(learning_rate)

    def fit(self, x, y, learning_rate=0.1, lambda_=0.01, num_iters=10000):
        """
        Parameters:
        -----------
        x : array_like
        y : array_like
        learning_rate : float
            Default : 0.1
        lambda_ : float
            Default : 0.0
        num_iters : int
            Default : 10000

        Returns:
        --------
        costs : List[floats]
        params : class[Parameters]
        """
        # Initialize parameters per layer
        params = {}
        for l in range(1, self.L + 1):
            params[l] = LinearParameters((self.nodes[l], self.nodes[l - 1]), self.b

        costs = []
        for i in range(num_iters):
            cache = self.forward_propagation(params, x)
            cost = self.cost_function(params, cache[’a’][self.L], y, lambda_)
            costs.append(cost)
            self.backward_propagation(params, cache, y)
            self.update_parameters(params, learning_rate)

            if i % 1000 == 0:
                print(f’Cost␣after␣iteration␣{i}:␣{cost}’)

        return params

    def evaluate(self, params, x):
        """
        Parameters:
        -----------
        params : class[Parameters]
        x : array_like

        Returns:
        --------
        y_hat : array_like
        """
        cache = self.forward_propagation(params, x)
        a = cache[’a’][self.L]
        y_hat = (~(a < 0.5)).astype(int)
        return y_hat

    def accuracy(self, params, x, y):
        """
        Parameters:
        -----------
        params : class[Parameters]
        x : array_like
        y : array_like

        Returns:
        --------
        accuracy : float
        """
        y_hat = self.evaluate(params, x)
        acc = np.sum(y_hat == y) / y.shape[1]

        return acc
```

## 3.3 Implementation in Python via `tensorflow`

We implement a neural network using `tensorflow.keras`.

```
#! python3

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from tensorflow import keras
from keras import Model, Input
from keras.layers import Dense

def keras_functional_nn(csv):
    df = pd.read_csv(csv)
    dataset = df.values
    x, y = dataset[:, :-1], dataset[:, -1].reshape(-1, 1)
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.15)
    train = {’x’ : x_train, ’y’ : y_train}
    test = {’x’ : x_test, ’y’ : y_test}
    mu = np.mean(train[’x’], axis=0, keepdims=True)
    var = np.var(train[’x’], axis=0, keepdims=True)
    train[’x’] = (train[’x’] - mu) / np.sqrt(var)
    test[’x’] = (test[’x’] - mu) / np.sqrt(var)

    ## Define network structure
    input_layer = Input(shape=(10,))
    hidden_layer_1 = Dense(
        32,
        activation=’relu’,
        kernel_initializer=’he_normal’,
        bias_initializer=’zeros’
    )(input_layer)
    hidden_layer_2 = Dense(
        8,
        activation=’relu’,
        kernel_initializer=’he_normal’,
        bias_initializer=’zeros’
    )(hidden_layer_1)
    output_layer = Dense(
        1,
        activation=’sigmoid’,
        kernel_initializer=’he_normal’,
        bias_initializer=’zeros’
    )(hidden_layer_2)

    model = Model(inputs=input_layer, outputs=output_layer)
    model.summary()

    ## Compile desired model
    model.compile(
        loss=’binary_crossentropy’,
        optimizer=’adam’,
        metrics=[’accuracy’]
    )

    ## Train the model
    hist = model.fit(
        train[’x’],
        train[’y’],
        batch_size=32,
        epochs=150,
        validation_split=0.17
    )

    ## Evaluate the model
    test_scores = model.evaluate(test[’x’], test[’y’], verbose=2)
    print(f’Test␣Loss:␣{test_scores[0]}’)
    print(f’Test␣Accuracy:␣{test_scores[1]}’)
```

## 3.4 Better Backpropagation

We consider a neural network of the form

$$\begin{tikzcd}
 & \mathbb{R}^{n^{[1]}\times n^{[0]}} \arrow[d] & \mathbb{R}^{n^{[1]}} \arrow[d] & & \mathbb{R}^{n^{[2]}\times n^{[1]}} \arrow[d] & \mathbb{R}^{n^{[2]}} \arrow[d] & & \mathbb{R}^{n^{[2]}} \arrow[d] & \\
\mathbb{R}^{n^{[0]}} \arrow[r] & \boxed{\phi^{[1]}} \arrow[r] & \boxed{\psi^{[1]}} \arrow[r] & \boxed{G^{[1]}} \arrow[r] & \boxed{\phi^{[2]}} \arrow[r] & \boxed{\psi^{[2]}} \arrow[r] & \boxed{G^{[2]}} \arrow[r] & \boxed{\mathbb{L}} \arrow[r] & \mathbb{R}
\end{tikzcd}$$

where we have the functions:

1. $$G^{[\ell]} : \mathbb{R}^{n^{[\ell]}} \to \mathbb{R}^{n^{[\ell]}}$$

is the broadcasting of the activation unit $g^{[\ell]} : \mathbb{R} \to \mathbb{R}$.

2.
$$\phi^{[\ell]} : \mathbb{R}^{n^{[\ell]} \times n^{[\ell-1]}} \times \mathbb{R}^{n^{[\ell-1]}} \to \mathbb{R}^{n^{[\ell]}}$$
is given by
$$\phi^{[\ell]}(W, x) = Wx.$$

3.
$$\psi^{[\ell]} : \mathbb{R}^{n^{[\ell]}} \times \mathbb{R}^{n^{[\ell]}} \to \mathbb{R}^{n^{[\ell]}}$$
is given by
$$\psi^{[\ell]}(b, x) = x + b.$$

4.
$$\mathbb{L} : \mathbb{R}^{n^{[2]}} \times \mathbb{R}^{n^{[2]}} \to \mathbb{R}$$
is the given loss-function.

We now consider back-propagating through the neural network via "reverse exterior differentiation". We represent our various reverse derivatives via the following diagram:

$$\begin{tikzcd}
 & \mathbb{R}^{n^{[1]} \times n^{[0]}} & \mathbb{R}^{n^{[1]}} & & \mathbb{R}^{n^{[2]} \times n^{[1]}} & \mathbb{R}^{n^{[2]}} & & \mathbb{R}^{n^{[2]}} \arrow[d] & \\
\mathbb{R}^{n^{[0]}} & \boxed{\phi^{[1]}} \arrow[l, "r"'] \arrow[u, "r_1"] & \boxed{\psi^{[1]}} \arrow[l, "r"'] \arrow[u, "\overline{r}_1"] & \boxed{G^{[1]}} \arrow[l, "r"'] & \boxed{\phi^{[2]}} \arrow[l, "r"'] \arrow[u, "r_2"] & \boxed{\psi^{[2]}} \arrow[l, "r"'] \arrow[u, "\overline{r}_2"] & \boxed{G^{[2]}} \arrow[l, "r"'] & \boxed{\mathbb{L}} \arrow[l, "r"'] & \mathbb{R} \arrow[l]
\end{tikzcd}$$

First, we need to consider our individual derivatives:

1. Suppose $G : \mathbb{R}^n \to \mathbb{R}^n$ is the broadcasting of $g : \mathbb{R} \to \mathbb{R}$. Then for $(x, \xi) \in T\mathbb{R}^n$, we have that
$$\begin{aligned} dG_x(\xi) &= G'(x) \odot \xi \\ &= \operatorname{diag}(G'(x)) \cdot \xi \end{aligned}$$
and for any $\zeta \in T_{G(x)}\mathbb{R}^n$, the reverse derivative is given by
$$\begin{aligned} rG_x(\zeta) &= G'(x) \odot \zeta \\ &= \operatorname{diag}(G'(x)) \cdot \zeta. \end{aligned}$$

2. Suppose $\phi : \mathbb{R}^{m \times n} \times \mathbb{R}^n \to \mathbb{R}^m$ is given by
$$\phi(A, x) = Ax.$$
Then we have two differentials to consider:

(a) For any $(A, x) \in \mathbb{R}^{m\times n} \times \mathbb{R}^n$ and any $\xi \in T_x\mathbb{R}^n$, we have that

$$\begin{aligned} d\phi_{(A,x)}(\xi) &= A\xi \\ &= L_A(\xi); \end{aligned}$$

and for any $\zeta \in T_{\phi(A,x)}\mathbb{R}^m$, we have the reverse derivative

$$\begin{aligned} r\phi_{(A,x)}(\zeta) &= A^T\zeta \\ &= L_{A^T}(\zeta); \end{aligned}$$

where $L_A(B) = AB$, i.e., left-multiplication by $A$.

(b) For any $(A, x) \in \mathbb{R}^{m\times n} \times \mathbb{R}^n$ and any $Z \in T_A\mathbb{R}^{m\times n}$ we have that

$$\begin{aligned} d_1\phi_{(A,x)}(Z) &= Zx \\ &= R_x(Z); \end{aligned}$$

and for any $\zeta \in T_{\phi(A,x)}\mathbb{R}^m$, we have the reverse derivative

$$\begin{aligned} r_1\phi_{(A,x)}(\zeta) &= \zeta x^T \\ &= R_{x^T}(\zeta); \end{aligned}$$

where $R_A(B) = BA$, i.e, right-multiplication by $A$.

3. Suppose $\psi : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$ is given by

$$\psi(b, x) = x + b.$$

Then we again have two (identical) differentials to consider:

(a) For any $(x, b) \in \mathbb{R}^n \times \mathbb{R}^n$ and any $\xi \in T_x\mathbb{R}^n$, we have that

$$d\psi_{(b,x)}(\xi) = \xi;$$

and for any $\zeta \in T_{\psi(b,x)}\mathbb{R}^n$, we have the reverse derivative

$$r\psi_{(b,x)}(\zeta) = \zeta.$$

(b) For any $(x, b) \in \mathbb{R}^n \times \mathbb{R}^n$ and any $\eta \in T_b\mathbb{R}^n$, we have that

$$d_1\psi_{(b,x)}(\eta) = \eta;$$

and for any $\zeta \in T_{(\psi(b,x)}\mathbb{R}^n$, we have the reverse derivative

$$\overline{r}_1\psi_{(b,x)}(\zeta) = \zeta.$$

**Proposition 3.1.** *Suppose we have the compositional diagram*

$$\mathbb{R}^n \xrightarrow{f} \mathbb{R}^m \xrightarrow{g} \mathbb{R}^k \xrightarrow{h} \mathbb{R}^l$$

*and we let* $F = h \circ g \circ f : \mathbb{R}^n \to \mathbb{R}^l$. *Then for any* $x \in \mathbb{R}^n$ *and any* $\zeta \in T_{F(x)}\mathbb{R}^l$, *the reverse derivative satisfies*

$$rF_x(\zeta) = rf_x \circ rg_{f(x)} \circ rh_{g(f(x)}(\zeta).$$

**Proof:** For any $\xi \in T_x\mathbb{R}^n$ and any $\zeta \in T_{F(x)}\mathbb{R}^l$, we have by definition

$$\begin{aligned}
\langle rF_x(\zeta), \xi \rangle_{\mathbb{R}^n} &= \langle \zeta, dF_x(\xi) \rangle_{\mathbb{R}^l} \\
&= \left\langle \zeta, dh_{g(f(x))} \circ dg_{f(x)} \circ df_x(\xi) \right\rangle_{\mathbb{R}^l} \\
&= \left\langle rh_{g(f(x))}(\zeta), dg_{f(x)} \circ df_x(\xi) \right\rangle_{\mathbb{R}^k} \\
&= \left\langle rg_{f(x)} \circ rh_{g(f(x))}(\zeta), df_x(\xi) \right\rangle_{\mathbb{R}^m} \\
&= \left\langle rf_x \circ rg_{f(x)} \circ rh_{g(f(x))}(\zeta), \xi \right\rangle_{\mathbb{R}^n}
\end{aligned}$$

as desired. □

**Lemma 3.2.** *Suppose* $f : \mathbb{R}^{n\times m} \to \mathbb{R}^k$, *and for* $P \in \mathbb{R}^{n\times m}$, *let* $R = rf_P$. *Then* $R \in \mathbb{R}^k{}_n{}^m$ *is rank* $(1, 2)$*-tensor written in coordinates as*

$$R = R_i{}^\mu{}_\nu \frac{\partial}{\partial X^\mu_\nu} \otimes dx^i,$$

*and the components is given by*

$$R_i{}^\mu{}_\nu = \frac{\partial f^i}{\partial X^\nu_\mu}$$

**Proof:** Considering the basis vectors $\frac{\partial}{\partial X^\nu_\mu} \in T_P\mathbb{R}^{n\times m}$ and $\frac{\partial}{\partial x^i} \in T_{f(P)}\mathbb{R}^k$ we have that

$$\begin{aligned}
R_i{}^\mu{}_\nu &= \left\langle R\left(\frac{\partial}{\partial x^i}\right), \frac{\partial}{\partial X^\nu_\mu} \right\rangle_F \\
&= \left\langle \frac{\partial}{\partial x^i}, df_P\left(\frac{\partial}{\partial X^\nu_\mu}\right) \right\rangle_{\mathbb{R}^k} \\
&= \left\langle \frac{\partial}{\partial x^i}, \frac{\partial f^\alpha}{\partial X^\nu_\mu}\frac{\partial}{\partial x^\alpha} \right\rangle_{\mathbb{R}^k} \\
&= \delta_{i\alpha}\frac{\partial f^\alpha}{\partial X^\nu_\mu},
\end{aligned}$$

as desired. □

Returning to our neural network, for each point $(x_j, y_j)$ in our training set, we first let

$$F_j := \mathbb{L} \circ G^{[2]} \circ \psi^{[2]} \circ \phi^{[2]} \circ G^{[1]} \circ \psi^{[1]} \circ \phi^{[1]},$$

and we have our cost function

$$\mathbb{J} := \frac{1}{N} \sum_{j=1}^{N} F_j.$$

We use the following notation for our inputs and outputs of our respective functions:

- $$\phi^{[\ell]} : (W^{[\ell]}, a^{[\ell-1]}{}_j) \mapsto u^{[\ell]}{}_j,$$
- $$\psi^{[\ell]} : (b^{[\ell]}, u^{[\ell]}{}_j) \mapsto v^{[\ell]}{}_j,$$
- $$G^{[\ell]} : v^{[\ell]}{}_j \mapsto a^{[\ell]}{}_j.$$

Let $p = (W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]})$ is a point in our parameter space. Suppose we wish to apply gradient descent with learning rate $\alpha \in T_{\mathbb{J}(p)}\mathbb{R}$, we would define our parameter updates via

$$\begin{aligned} W^{[1]} &:= W^{[1]} - r_1\mathbb{J}_p(\alpha) \\ b^{[1]} &:= b^{[1]} - \overline{r}_1\mathbb{J}_p(\alpha) \\ W^{[2]} &:= W^{[2]} - r_2\mathbb{J}_p(\alpha) \\ b^{[2]} &:= b^{[2]} - \overline{r}_2\mathbb{J}_p(\alpha). \end{aligned}$$

Moreover, by linearity (and independence of our training data), we see that

$$r\mathbb{J}_p = \frac{1}{N} \sum_{j=1}^{N} r(F_j)_p,$$

so we need only calculate the various reverse derivatives of $F_j$.

To this end, we suppress the index $j$ when we're working with the compositional function $F$. We calculate the reverse derivatives in the order traversed in our back-propagating path along the network.

1. $\overline{r}_2\mathbb{J}_p$:

$$\begin{aligned}\overline{r}_2F_p &= \overline{r}_2(\mathbb{L}\circ G^{[2]}\circ\psi^{[2]})_p\\ &= \overline{r}_2\psi^{[2]}_p\circ rG^{[2]}_{v^{[2]}}\circ r\mathbb{L}_{a^{[2]}}\\ &= \mathbb{1}\circ rG^{[2]}_{v^{[2]}}\circ r\mathbb{L}_{a^{[2]}}\\ &= rG^{[2]}_{v^{[2]}}\circ r\mathbb{L}_{a^{[2]}},\end{aligned}$$

   and hence

$$\overline{r}_2\mathbb{J}_p = \frac{1}{N}\sum_{j=1}^{N} rG^{[2]}_{v^{[2]}{}_j}\cdot r\mathbb{L}_{a^{[2]}{}_j}$$

2. $r_2\mathbb{J}_p$:

$$\begin{aligned}r_2F_p &= r_2(\mathbb{L}\circ G^{[2]}\circ\psi^{[2]}\circ\phi^{[2]})_p\\ &= r_2\phi^{[2]}_p\circ r\psi^{[2]}_{u^{[2]}}\circ rG^{[2]}_{v^{[2]}}\circ r\mathbb{L}_{a^{[2]}}\\ &= R_{a^{[1]T}}\circ\mathbb{1}\circ rG^{[2]}_{v^{[2]}}\circ r\mathbb{L}_{a^{[2]}}\\ &= R_{a^{[1]T}}\circ rG^{[2]}_{v^{[2]}}\circ r\mathbb{L}_{a^{[2]}},\end{aligned}$$

   and hence

$$r_2\mathbb{J}_p = \frac{1}{N}\sum_{j=1}^{N} R_{a^{[1]T}{}_j}\circ rG^{[2]}_{v^{[2]}{}_j}\cdot r\mathbb{L}_{a^{[2]}{}_j}.$$

   Notice that this is not just a sum after matrix multiplication since we have composition with an operator, namely, $R_{a^{[1]T}{}_j}$. However, since the learning rate $\alpha\in T_{\mathbb{J}(p)}\mathbb{R}\cong\mathbb{R}$, which may pass through the aforementioned linear composition, we conclude that

$$\begin{aligned}r_2\mathbb{J}_p &= \frac{1}{N}\sum_{j=1}^{N} R_{a^{[1]T}{}_j}\circ rG^{[2]}_{v^{[2]}{}_j}\cdot r\mathbb{L}_{a^{[2]}{}_j}\\ &= \frac{1}{N}\sum_{j=1}^{N} rG^{[2]}_{v^{[2]}{}_j}\cdot r\mathbb{L}_{a^{[2]}{}_j}a^{[1]T}{}_j.\end{aligned}$$

3. $\overline{r}_1\mathbb{J}_p$:

$$\begin{aligned}\overline{r}_1F_p &= \overline{r}_1(\mathbb{L}\circ G^{[2]}\circ\psi^{[2]}\circ\phi^{[2]}\circ G^{[1]}\circ\psi^{[1]})_p\\ &= \overline{r}_1\psi^{[1]}_p\circ rG^{[1]}_{v^{[1]}}\circ r\phi^{[2]}_{a^{[1]}}\circ r\psi^{[2]}_{u^{[2]}}\circ rG^{[2]}_{v^{[2]}}\circ r\mathbb{L}_{a^{[2]}}\\ &= \mathbb{1}\circ rG^{[1]}_{v^{[1]}}\circ L_{W^{[2]T}}\circ\mathbb{1}\circ rG^{[2]}_{v^{[2]}}\circ r\mathbb{L}_{a^{[2]}}\\ &= rG^{[1]}_{v^{[1]}}\circ L_{W^{[2]T}}\circ rG^{[2]}_{v^{[2]}}\circ r\mathbb{L}_{a^{[2]}},\end{aligned}$$

and hence

$$\overline{r}_1\mathbb{J}_p = \frac{1}{N}\sum_{j=1}^{N} rG^{[1]}_{v^{[1]}_j} \cdot W^{[2]T} \cdot rG^{[2]}_{v^{[2]}_j} \cdot r\mathbb{L}_{a^{[2]}_j}.$$

4. $r_1\mathbb{J}_p$:

$$\begin{aligned} r_1F_p &= r_1(\mathbb{L} \circ G^{[2]} \circ \psi^{[2]} \circ \phi^{[2]} \circ G^{[1]} \circ \psi^{[1]} \circ \phi^{[1]})_p \\ &= r_1\phi^{[1]}_p \circ r\psi^{[1]}_{u^{[1]}} \circ rG^{[1]}_{v^{[1]}} \circ r\phi^{[2]}_{a^{[1]}} \circ r\psi^{[2]}_{u^{[2]}} \circ rG^{[2]}_{v^{[2]}} \circ r\mathbb{L}_{a^{[2]}} \\ &= R_{x^T} \circ \mathbb{1} \circ rG^{[1]}_{v^{[1]}} \circ L_{W^{[2]T}} \circ \mathbb{1} \circ rG^{[2]}_{v^{[2]}} \circ r\mathbb{L}_{a^{[2]}} \\ &= R_{x^T} \circ rG^{[1]}_{v^{[1]}} \circ L_{W^{[2]T}} \circ rG^{[2]}_{v^{[2]}} \circ r\mathbb{L}_{a^{[2]}}, \end{aligned}$$

and hence

$$\begin{aligned} r_1\mathbb{J}_p &= \frac{1}{N}\sum_{j=1}^{N} R_{x^T} \circ rG^{[1]}_{v^{[1]}} \cdot W^{[2]T} \cdot rG^{[2]}_{v^{[2]}} \cdot r\mathbb{L}_{a^{[2]}} \\ &= \frac{1}{N}\sum_{j=1}^{N} rG^{[1]}_{v^{[1]}} \cdot W^{[2]T} \cdot rG^{[2]}_{v^{[2]}} \cdot r\mathbb{L}_{a^{[2]}} \cdot x^T \end{aligned}$$

Part II

# Improving Deep Neural Networks: Hyperparameter Tuning, Regularization, and Optimization

# 4 Training, Development and Test Sets

Let $\mathbb{D} = \{(x_j, y_j) \in \mathbb{R}^m \times \mathbb{R}^K : 1 \leq j \leq N\}$ denote a dataset. Then we partition $\mathbb{D}$ into three distinct sets

$$\mathbb{D} = \mathbb{X} + \mathcal{D} + \mathcal{T},$$

where $\mathbb{X}$ is called our *training set*, $\mathcal{D}$ is called our *development, or cross-validation set*, and $\mathcal{T}$ is called our *test set*. We make this partition randomly, however, if $N = |\mathbb{D}| \leq 10^4$, we see a partition being divided accordingly to the following ratios:

$$n_X := |\mathbb{X}| \approx \frac{3}{5}N,$$

$$n_D := |\mathcal{D}| \approx \frac{1}{5}N,$$

and

$$n_T := |\mathfrak{T}| \approx \frac{1}{5}N.$$

If however, we have a very large dataset (i.e., $N > 10^4$), then we assume a much smaller ratio of something similar to

$$\frac{n_X}{N} \approx 0.98, \qquad \frac{n_D}{N} \approx 0.01, \qquad \frac{n_T}{N} \approx 0.01.$$

In general, we use our training set $\mathbb{X}$ to train our parameters $W^{[\ell]}$ and $b^{[\ell]}$, we use our development set $\mathcal{D}$ to tune our hyper-parameters (i.e., learning rate, number of layers, number of nodes per layer, activation function, number of iterations to perform gradient descent, regularization parameters, etc), and we use our test set $\mathcal{T}$ to evaluate the accuracy of our model. Since we're partitioning our dataset to better increase the accuracy of our model, we need to define an error function. To this end, define $\mathcal{E} : 2^{\mathbb{D}} \to [0, 1]$ by

$$\mathcal{E}(\mathcal{A}) = \frac{1}{|\mathcal{A}|} \sum_{(x,y) \in \mathcal{A}} \varepsilon(x, y),$$

where $\varepsilon : \mathbb{D} \to \{0, 1\}$ is defined by

$$\varepsilon(x, y) = \begin{cases} 1 & \text{if } y = \hat{y}(x) \\ 0 & \text{else.} \end{cases}$$

From our partition and error function we can make several claims of the fitting of our model to our data. Indeed, let $\epsilon > 0$ be a small percentage (with exact value depending on specific examples), then:

- If $\mathcal{E}(\mathbb{X}) < \epsilon$ and $\mathcal{E}(\mathbb{X}) < \mathcal{E}(\mathcal{D}) \lessapprox 10\epsilon$, then we say our model has *high variance* since our model is overfitting the data.

- If $\mathcal{E}(\mathbb{X}) \approx \mathcal{E}(\mathcal{D}) \gtrapprox 10\epsilon$, then we say our model has *high bias* since our model is underfitting the data.

- If $10\epsilon \lessapprox \mathcal{E}(\mathbb{X}) \ll \mathcal{E}(\mathcal{D})$, then we say our model has both high bias (since it doesn't fit our training data well) and high variance (because the model fits the training data better than the development data).

- If $\mathcal{E}(\mathbb{X}), \mathcal{E}(\mathcal{D}) < \epsilon$, then we say the model has both low bias and low variance.

**Remark 4.1.** *The interpretations of our error percentage is based on two crucial assumptions:*

- *$\mathcal{D}$ and $\mathcal{T}$ come from samplings with the same distribution of outputs (i.e., if we're determining whether a collection of images contain a cat, we should never have that $\mathcal{D}$ is mostly cat pictures, and $\mathcal{T}$ is mostly non-cat pictures).*

- *The optimal error for the model is approximately* $0\%$*. That is, if a human were looking at the data, they could determine the correct response with negligible error. This is sometimes called the Bayes error.*

*If either of these assumptions fail to hold, other methods of analysis may be required to obtain meaningful insights for the performance of our model.*

A methodology for using errors could be as follows

1. Check $\mathcal{E}(\mathbb{X})$ for high bias.

   a. If "Yes", then we can try a bigger network, we can train longer, or we can change the neural network architecture. Then we return to (1.).

   b. If "No", then we move to (2.).

2. Check $\mathcal{E}(\mathcal{D})$ for high variance.

   a. If "Yes", then we can try to get more data, try regularization, or try changing the neural network architecture. Then we return to (1.).

   b. If "No", then we're done.

## 4.1 Python Implementation

To implement a partitioning we could do something like the following:

```python
import numpy as np
from sklearn.utils import shuffle

def partition_data(x, y, train_ratio):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (m, N)
    y : array_like
        y.shape = (k, N)
    train_ratio : float
        0<=train_ratio<=1

    Returns
    -------
    train : Tuple[array_like]
    dev : Tuple[array_like]
    test : Tuple[array_like]
    """
    ## Shuffle the data
    x, y = shuffle(x.T, y.T) #
    x = x.T
    y = y.T

    ## Get the size of partitions
    N = x.shape[1]
    N_train = int(train_ratio * N)
    N_mid = (N - N_train) // 2

    ## Create partitions
    train = (x[:,:N_train], y[:,:N_train])
    dev = (x[:,N_train:N_train+N_mid], y[:,N_train:N_train+N_mid])
    test = (x[:,N_train+N_mid:], y[:,N_train+N_mid:])

    assert(x.all() == np.concatenate([train[0], dev[0], test[0]], axis=1).all())
    assert(y.all() == np.concatenate([train[1], dev[1], test[1]], axis=1).all())

    return train, dev, test
```

# 5 Regularization

Suppose we're training an $L$-layer neural network with dataset $\{(x_j, y_j)\} \subset \mathbb{R}^{m_0} \times \mathbb{R}^{m_L}$ with $N$ examples. Assuming a generic loss function $\mathbb{L} : \mathbb{R}^{m_L} \times \mathbb{R}^{m_L} \to \mathbb{R}$, then we have our cost function $\mathbb{J}$ defined on our one-parameter families of parameters $W$ and $b$ given by

$$\mathbb{J}(W, b) = \frac{1}{N} \sum_{j=1}^{N} \mathbb{L}(\hat{y}_j, y_j).$$

If our model suffers from overfitting the training set, it's reasonable to impose constraints on the parameters $W$ and/or $b$. That is, define the function

$$R(W) = \frac{\lambda}{2N} \sum_{\ell=1}^{L} \left\| W^{[\ell]} \right\|_F^2,$$

for some $\lambda > 0$, where $\|\cdot\|_F$ represents the Frobenius norm on matrices, and we define the *regularized cost function* $\mathbb{J}^R$ given by

$$\begin{aligned} \mathbb{J}^R(W, b) &= \mathbb{J}(W, b) + R(W) \\ &= \frac{1}{N} \sum_{j=1}^{N} \mathbb{L}(\hat{y}_j, y_j) + \frac{\lambda}{2N} \sum_{\ell=1}^{L} \left\| W^{[\ell]} \right\|_F^2. \end{aligned}$$

Adding such an $R(W)$ to our cost function is known as $L^2$*-regularization.* We note that by linearity we have the following equalities amongst gradients:

$$\frac{\partial \mathbb{J}^R}{\partial b^{[\ell]}} = \frac{\partial \mathbb{J}}{\partial b^{[\ell]}}$$

and

$$\frac{\partial \mathbb{J}^R}{\partial W^{[\ell]}} = \frac{\partial \mathbb{J}}{\partial W^{[\ell]}} + \frac{\lambda}{N} W^{[\ell]}.$$

The idea behind regularization is that we're now minimizing

$$\min_{W,b} \mathbb{J}^R(W, b) = \min_{W,b} \left\{ \mathbb{J}(W, b) + R(W) \right\},$$

and so for suitably chosen $\lambda > 0$, it forces $\left\| W^{[\ell]} \right\|_F$ to be small, along with minimizing the cost $\mathbb{J}$. This balancing-act of minimizing the two functions simultaneously helps with overfitting the data.

A typical usage of regularization would be similar to the following outline:

i. Partition our dataset $\mathbb{D} = \mathbb{X} \cup \mathcal{D} \cup \mathcal{T}$.

ii. Give a set $\Lambda$ of potential regularization parameters.

iii. For each $\lambda \in \Lambda$, we first train on $\mathbb{X}$, that is, we obtain

$$\begin{aligned}(W, b) &= \arg\min_{W,b} \mathbb{J}^R(W, b) \\ &= \arg\min_{W,b} \left\{ \frac{1}{n_X} \sum_{(x,y)\in\mathbb{X}} \mathbb{L}(\hat{y}, y) + \frac{\lambda}{2n_X} \sum_{\ell=1}^{L} \left\| W^{[\ell]} \right\|_F^2 \right\}\end{aligned}$$

which dependent on $\lambda$.

iv. Then using the aforementioned $(W, b) = (W, b)(\lambda)$, we evaluate $\mathcal{E}_\lambda(\mathbb{X})$ and $\mathcal{E}_\lambda(\mathcal{D})$.

v. After finding $\mathcal{E}_\lambda(\mathbb{X})$ and $\mathcal{E}_\lambda(\mathcal{D})$ for each $\lambda \in \Lambda$, we choose our desired $\lambda$ and hence our desired parameters $W$ and $b$.

vi. We evaluate our model on $\mathcal{T}$ to determine the overall accuracy.

## 5.1 Python Implementation

```
import numpy as np

import mlLib.utils as utils
import mlLib.npActivators as npActivators

def forward_propagation(x, params, activators):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0] n)
    params : Dict[Dict]
        params[’w’][l] : array_like
            wl.shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            bl.shape = (layers[l], 1)
    activators : List[str]
        activators[l] = activation function of layer l+1
    Returns
    -------
    cache : Dict[Dict]
```

```
        cache[’z’][l] : array_like
            z[l].shape = (layers[l], n)
        cache[’a’][l] : array_like
            a[l].shape = (layers[l], n)
    """
    # Retrieve parameters
    w = params[’w’]
    b = params[’b’]
    L = len(w) # Number of layers excluding output layer
    n = x.shape[1]
    # Set empty caches
    a = {}
    z = {}
    # Initialize a
    a[0] = x
    for l in range(1, L + 1):
        z[l], a[l] = utils.linear_activation_forward(a[l - 1], w[l], b[l], activator

    cache = {’a’ : a, ’z’ : z}
    return cache

def compute_cost(y, params, cache, lambda_=0.0):
    """
    Parameters
    ----------
    y : array_like
        y.shape = (layers[-1], n)
    params : Dict[Dict[array_like]]
        params[’w’][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            b[l].shape = (layers[l], 1)
    cache : Dict[Dict[array_like]]
        cache[’z’][l] : array_like
            z[l].shape = (layers[l], n)
        cache[’a’][l] : array_like
            a[l].shape = (layers[l], n)
    lambda_ : float
        Default: 0.0

    Returns
    -------
    cost : float
        The cost evaluated at y and aL
    """
    ## Retrieve parameters
    n = y.shape[1]
    a = cache[’a’]
    w = params[’w’]
    L = len(a)
    aL = a[L - 1]

    ## Regularization term
    R = 0
    for l in range(1, L):
        R += np.sum(w[l] * w[l])
    R *= (lambda_ / (2 * n))

    ## Unregularized cost
    J = (-1 / n) * (np.sum(y * np.log(aL)) + np.sum((1 - y) * np.log(1 - aL)))

    ## Total Cost
    cost = J + R
    cost = float(np.squeeze(cost))
    return cost

def backward_propagation(x, y, params, cache, activators, lambda_=0.0):
    ”””
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[-1], n)
    params : Dict[Dict[array_like]]
        params[’w’][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            b[l].shape = (layers[l], 1)
    cache : Dict[Dict[array_like]]
        cache[’a’][l] : array_like
            a[l].shape = (layers[l], n)
        cache[’z’][l] : array_like
            z[l].shape = (layers[l], n)
    activators : List[str]
        activators[l] = activation function of layer l+1
    lambda_ : float
        Default: 0.0

    Returns
    -------
    grads : Dict[Dict]
        grads[’dw’][l] : array_like
            dw[l].shape = w[l].shape
        grads[’db’][l] : array_like
            db[l].shape = b[l].shape
    """
    ## Retrieve parameters
    a = cache[’a’]
    z = cache[’z’]
    w = params[’w’]
    n = x.shape[1]
    L = len(z)

    ## Compute deltas
    delta = {}
    delta[L] = a[L] - y
    for l in reversed(range(1, L)):
        delta[l] = utils.linear_activation_backward(delta[l + 1], z[l], w[l + 1], ac

    ## Compute gradients
    dw = {}
    db = {}
    for l in range(1, L + 1):
        db[l] = (1 / n) * np.sum(delta[l], axis=1, keepdims=True)
        assert(db[l].shape == (w[l].shape[0], 1))
        dw[l] = (1 / n) * (delta[l] @ a[l - 1].T + lambda_ * w[l])
        assert(dw[l].shape == w[l].shape)
    grads ={’dw’ : dw, ’db’ : db}
    return grads


def model(x, y,
          hidden_layer_sizes,
          activators,
          lambda_=0.0,
          num_iters=1e4,
          print_cost=False):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[-1], n)
    hidden_layer_sizes : List[int]
        The number nodes layer l = hidden_layer_sizes[l-1]
    activators : List[str]
        activators[l] = activation function of layer l+1
    lambda_ : float
        The regularization parameter
        Default: 0.0
    num_iters : int
        Number of iterations with which our model performs gradient descent
        Default: 10000
    print_cost : Boolean
        If True, print the cost every 1000 iterations
        Default: False

    Returns
    -------
    params : Dict[Dict]
        params[’w’][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            b[l].shape = (layers[l], 1)
    cost : float
        The final cost value for the optimized parameters returned
    ”””
    ## Set dimensions and Initialize parameters
    n, layers = utils.dim_retrieval(x, y, hidden_layer_sizes)
    params = utils.initialize_parameters_random(layers)

    # main gradient descent loop
    for i in range(num_iters):
        cache = forward_propagation(x, params, activators)
        cost = compute_cost(y, params, cache, lambda_)
        grads = backward_propagation(x, y, params, cache, activators, lambda_)
        params = utils.update_parameters(params, grads)

        if print_cost and i % 1000 == 0:
            print(f’Cost␣after␣iteration␣{i}:␣{cost}’)

    return params, cost
```

## 5.2 (Inverted) Dropout Regularization

For illustrative purposes, suppose we have a 3-layer neural network of the following form:

$$\underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{m_0} \end{bmatrix}}_{\text{Layer 0}} \xrightarrow{\varphi^{[1]}} \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]m_1} \end{bmatrix} \xrightarrow{g^{[1]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]m_1} \end{bmatrix}}_{\text{Layer 1}} \xrightarrow{\varphi^{[2]}} \underbrace{\begin{bmatrix} z^{[2]1} \\ \vdots \\ z^{[2]m_2} \end{bmatrix} \xrightarrow{g^{[2]}} \begin{bmatrix} a^{[2]1} \\ \vdots \\ a^{[2]m_2} \end{bmatrix}}_{\text{Layer 2}} \xrightarrow{\varphi^{[3]}} \text{output},$$

Let $Q_0, Q_1, Q_2$ denote the collection of all nodes in Layers $0, 1, 2$, respectively. Let $p_0, p_1, p_2 \in [0, 1]$, and define a probability distribution $\mathbb{P}_\ell$ on $Q_\ell$ by

$$\mathbb{P}_\ell(q = 1) = p_\ell, \qquad \mathbb{P}_\ell(q = 0) = 1 - p_\ell,$$

where $q = 1$ represents the node existing in layer-$\ell$, and $q = 0$ represents the dropping of the node from layer-$\ell$. That is we're effectively reducing the number of nodes throughout the network, thus simplifying the network and reducing the amount of influence of any single feature or node on the entire model. That is, we would implement a methodology similar to the following:

i. For each layer $\ell$ and each training example $x_j$ define the "dropout vector" ${D^{[\ell]}}_j$ by

$${D^{[\ell]}}_j = \begin{bmatrix} d_j^1 \\ \vdots \\ d_j^{m_\ell} \end{bmatrix},$$

where

$$d_j^i = \begin{cases} 1 & \text{if } \mathbb{P}(q^i) \leq p_\ell \\ 0 & \text{if } \mathbb{P}(q^i) > p_\ell \end{cases}.$$

ii. During forward propagation, we redefine

$$a^{[\ell]} \mapsto \frac{a^{[\ell]} \odot D^{[\ell]}}{p_\ell}.$$

iii. During backward propagation, we define

$$\delta^{[\ell]} \mapsto \frac{\delta^{[\ell]} \odot D^{[\ell]}}{p_\ell}.$$

iv. Then perform gradient descent, etc with these new values.

### 5.2.1 Python Implementation

We see here the use of inverted dropout regularization in a general neural network.

```python
import numpy as np

import mlLib.utils as utils

def dropout_matrices(layers, num_examples, keep_prob):
    """
    Parameters
    ----------
    layers : List[int]
        layers[l] = number of nodes in layer l
    num_examples : int
        The number of training examples
    keep_prob : List[float]
        keep_prob[l] = The probabilty of keeping a node in layer l

    Returns
    -------
    D : Dict[array_like]
        D[l].shape = (layers[l], num_ex)
        D[l] = a Boolean array
    """
    np.random.seed(1)
    L = len(layers)
    D = {}
    for l in range(L - 1):
        D[l] = np.random.rand(layers[l], num_examples)
        D[l] = (D[l] < keep_prob[l]).astype(int)
        assert(D[l].shape == (layers[l], num_examples))
    return D



def forward_propagation(x, params, activators, D, keep_prob):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0] n)
    params : Dict[Dict]
        params[’w’][l] : array_like
            wl.shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            bl.shape = (layers[l], 1)
    activators : List[str]
        activators[l] = activation function of layer l+1
    D : Dict[array_like]
        D[l].shape = (layer_dims[l], num_ex)
        D[l] = a Boolean array
    keep_prob : List[float]
        keep_prob[l] = The probabilty of keeping a node in layer l

    Returns
    -------
    cache : Dict[Dict]
        cache[’z’][l] : array_like
            z[l].shape = (layers[l], n)
        cache[’a’][l] : array_like
            a[l].shape = (layers[l], n)
    """
    # Retrieve parameters
    w = params[’w’]
    b = params[’b’]
    L = len(w) # Number of layers including input layer
    n = x.shape[1]

    # Set empty caches
    a = {}
    z = {}
    # Dropout on layer 0
    a[0] = x
    a[0] = a[0] * D[0]
    a[0] /= keep_prob[0]
    # Loop through hidden layers
    for l in range(1, L):
        zl, al = utils.linear_activation_forward(a[l - 1], w[l], b[l], activators[l
        al = al * D[l]
        al /= keep_prob[l]
        z[l] = zl
        a[l] = al

    # Output layer
    z[L], a[L] = utils.linear_activation_forward(a[L - 1], w[L], b[L], activators[-

    cache = {’z’ : z, ’a’ : a}
    return cache

def backward_propagation(x, y, params, cache, activators, D, keep_prob):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[-1], n)
```

```
params : Dict
    params['w'][l] : array_like
        w[l].shape = (layers[l], layers[l-1])
    params['b'][l] : array_like
        b[l].shape = (layers[l], 1)
cache : Dict
    cache['a'][l] : array_like
        a[l].shape = (layers[l], n)
    cache['z'][l] : array_like
        z[l].shape = (layers[l], n)
activators : List[str]
    activators[l] = activation function of layer l+1
D : Dict[array_like]
    D[l].shape = (layer[l], num_ex)
    D[l] = a Boolean array
keep_prob : List[float]
    keep_prob[l] = The probabilty of keeping a node in layer l

Returns
-------
grads : Dict[Dict]
    grads['dw'][l] : array_like
        dw[l].shape = w[l].shape
    grads['db'][l] : array_like
        db[l].shape = b[l].shape
"""
## Retrieve parameters
a = cache['a']
z = cache['z']
w = params['w']
n = x.shape[1]
L = len(z)

## Compute deltas
delta = {}
delta[L] = a[L] - y
for l in reversed(range(1, L)):
    deltal = utils.linear_activation_backward(delta[l + 1], z[l], w[l + 1], act
    deltal = deltal * D[l]
    deltal /= keep_prob[l]
    delta[l] = deltal

## Compute gradients
dw = {}
db = {}

for l in range(1, L + 1):
        db[l] = (1 / n) * np.sum(delta[l], axis=1, keepdims=True)
        assert(db[l].shape == (w[l].shape[0], 1))
        dw[l] = (1 / n) * delta[l] @ a[l - 1].T
        assert(dw[l].shape == w[l].shape)
    grads = {’dw’ : dw, ’db’ : db}
    return grads

def model(x, y,
            hidden_sizes,
            activators,
            keep_prob = 1.0,
            num_iters=2500,
            learning_rate=0.1,
            print_cost=False):
    """
    Parameters
    ----------
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[-1], n)
    hidden_sizes : List[int]
        The number nodes layer l = hidden_sizes[l-1]
    activators : List[function]
        activators[l] = activation function of layer l+1
    keep_prob : List[float] | float
        keep_prob[l] = The probabilty of keeping a node in layer l
        keep_prob = The same probability for all input and hidden layers
    num_iters : int
        Number of iterations with which our model performs gradient descent
    learning_rate : float
        The learning rate for gradient descent
    print_cost : Boolean
        If True, print the cost every 1000 iterations

    Returns
    -------
    params : Dict[Dict]
        params[’w’][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            b[l].shape = (layers[l], 1)
    cost : float
        The final cost value for the optimized parameters returned
    """
    ## Retrieve parameters
    n, layers = utils.dim_retrieval(x, y, hidden_sizes)
    params = utils.initialize_parameters_random(layers)

    ## Expand keep_prob to a list if it's a single float
    if isinstance(keep_prob, float):
        keep_prob = [keep_prob] * (len(layers) - 1)
    ## Main gradient descent loop
    for i in range(num_iters):
        D = dropout_matrices(layers, n, keep_prob)
        cache = forward_propagation(x, params, activators, D, keep_prob)
        cost = utils.compute_cost(y, cache)
        grads = backward_propagation(x, y, params, cache, activators, D, keep_prob)
        params = utils.update_parameters(params, grads, learning_rate)

        if print_cost and i % 1000 == 0:
            print(f'Cost␣after␣iteration␣{i}:␣{cost}')

    return params, cost
```

## 5.3 Data Augmentation

This section requires work.

There are few other regularization techniques. One of the simplest techniques is data augmentation, i.e., transforming data you currently have into related but different example to gather a larger dataset (e.g., flipping or distorting images to obtain other relevant images).

## 5.4 Early Stopping

This section requires work.

Another technique is stop the training early (fewer iterations) before the model develops higher variance.

# 6 Gradients and Numerical Remarks

This section requires work. See “He Initialization” and “Xavier Initialization”

We first remark, that by our use of gradient descent, there are few outlier cases which may occur. Namely our gradients may explode or vanish. One way to attempt to fix such a situation to impose a normalization on our weights depending on our activation functions.

- If $g^{[\ell]} = \text{ReLU}$, then we wish to impose the requirement that

$$\mathbb{E}[(W^{[\ell]2})] = \frac{1}{m_{\ell-1}}.$$

## 6.1 Numerical Gradient Checking

Suppose $f : \mathbb{R}^n \to \mathbb{R}$ is a smooth function. Then, we recall the definition of the partial derivative

$$\begin{aligned}\frac{\partial f}{\partial x^j} &= \lim_{h\to 0} \frac{f(x+he_j) - f(x)}{h} \\ &= \lim_{\epsilon\to 0^+} \frac{f(x+\epsilon e_j) - f(x-\epsilon e_j)}{2\epsilon},\end{aligned}$$

and so for sufficiently small $\epsilon > 0$, we have the approximation

$$\frac{\partial f}{\partial x^j} \approx \frac{f(x+\epsilon e_j) - f(x-\epsilon e_j)}{2\epsilon}.$$

Define the approximation function $F : \mathbb{R}^n \times (0,1) \to \mathbb{R}^n$ by

$$F(x,\epsilon) = \frac{1}{2\epsilon}\begin{bmatrix} f(x+\epsilon e_1) - f(x-\epsilon e_1) \\ \vdots \\ f(x+\epsilon e_n) - f(x-\epsilon e_n)\end{bmatrix}.$$

Then we may check that our gradient computation $\nabla f(x)$ is correct by checking that

$$\frac{\|F(x,\epsilon) - \nabla f(x)\|_2}{\|F(x,\epsilon)\|_2 + \|\nabla f(x)\|_2} \approx 0.$$

## 6.2 Python Implementation

```python
## f(x) = x_1*x_2*...*x_n
def fctn(x):
    n = x.shape[0]
    y = np.prod(x)
    grad = np.zeros((n, 1))
    for i in range(n):
        omit = 1 - np.eye(1, n, i).T
        omit = np.array(omit, dtype=bool)
        grad[i, 0] = np.prod(x, where=omit)
    return y, grad

def gradient_check(grad, f, x, epsilon=1e-3):
    """
    Parameters
    ----------
    grad : array_like
        grad.shape= (n, 1)
    f : function
        The function to check.
    x : array_like
        x.shape = (n, 1)
    epsilon : float
        Default 0.001
    Returns
    error : float
    -------
    """
    n = x.shape[0]
    y_diffs = []
    for i in range(n):
        e = np.eye(1, n, i).T
        x_plus = x + epsilon * e
        x_minus = x - epsilon * e
        y_plus, _ = f(x_plus)
        y_minus, _ = f(x_minus)
        y_diffs.append(y_plus - y_minus)
    y_diffs = np.array(y_diffs).reshape(n, 1)
    y_diffs = y_diffs / (2 * epsilon)

    error = (np.linalg.norm(y_diffs - grad)
                / (np.linalg.norm(y_diffs) + np.linalg.norm(grad)))
    return error
```

# 7 Gradient Descent

So far in our implementation of gradient descent, we use the entire training set for every iteration of gradient descent. This method is called *batch gradient descent.* Gradient descent has many downfalls. Indeed, since we're typically working in a *very* high dimensional space, the majority of the critical points for our cost function are actually saddle points (these can be thought of as plateaus of the loss-manifold). These pitfalls (amongst others) are what we wish to overcome. To this end, we first consider a modification of batch gradient descent by partitioning the training set into smaller "mini-batches" and using each mini-batch recursively throughout the iterative process.

That is, suppose we have training set $\mathbb{X}$ with $|\mathbb{X}| = n$, where $n$ is very large (e.g., $n = 5000000$). We fix a batch size $b$ (e.g., $b = 5000$), and partition $\mathbb{X}$ into (e.g., 1000 distinct) mini-batches

$$\left\{\mathbb{X}^k : 1 \leq k \leq \left\lceil \frac{n}{b} \right\rceil\right\}, \qquad \mathbb{X} = \bigcup_{k=1}^{\lceil \frac{n}{b} \rceil} \mathbb{X}^k,$$

where $\lceil \frac{n}{b} \rceil$ denote the ceiling function. If we shuffle $\mathbb{X}$ and partition during each epoch (i.e., each iteration) so our loss-manifold changes during each batch iteration within each epoch, we can then perform gradient descent in the following manner:

1. For $0 \leq i <$ `num_iters`:

    a. Let $B = \lceil \frac{n}{b} \rceil$, and generate batches $\{\mathbb{X}^k\}$.

    b. For $1 \leq k \leq B$:

        i. Perform forward propagation on $\mathbb{X}^k$:

$$\begin{aligned} a^{[0]} &= x(\mathbb{X}^k) \\ z^{[\ell]} &= W^{[\ell]} a^{[\ell-1]} + b^{[\ell]} \\ a^{[\ell]} &= g^{[\ell]}(z^{[\ell]}) \end{aligned}$$

        ii. Evaluate the cost $\mathbb{J}^k$ on $\mathbb{X}^k$:

$$\mathbb{J}^k(W, b) = \frac{1}{|\mathbb{X}^k|} \sum_{(x,y)\in\mathbb{X}^k} \mathbb{L}(\hat{y}, y) + \frac{\lambda}{2|\mathbb{X}^k|} \sum_{\ell=1}^{L} \left\|W^{[\ell]}\right\|_F^2 .$$

iii. Perform backward propagation on $\mathbb{X}^k$:

$$\begin{aligned} \frac{\partial \mathbb{J}^k}{\partial W^{[\ell]}} &= \frac{1}{|\mathbb{X}^k|}\delta^{[\ell]}a^{[\ell-1]T} + \frac{\lambda}{|\mathbb{X}^k|}W^{[\ell]} \\ \frac{\partial \mathbb{J}^k}{\partial b^{[\ell]}} &= \frac{1}{|\mathbb{X}^k|}\sum_{\rho\sim\mathbb{X}^k} \delta^{[\ell]}{}_{\rho} \end{aligned}$$

iv. Perform gradient descent:

$$\begin{aligned} W^{[\ell]} &:= W^{[\ell]} - \alpha\frac{\partial \mathbb{J}^k}{\partial W^{[\ell]}} \\ b^{[\ell]} &:= b^{[\ell]} - \alpha\frac{\partial \mathbb{J}^k}{\partial b^{[\ell]}} \end{aligned}$$

We make several remarks about mini-batch gradient descent:

- Batch gradient descent doesn't always decrease (e.g., our learning rate is too large). Mini-batch may oscillate rapidly, but the general direction should move towards a minimum.

- If $b = n$, then we fully recover batch gradient descent. This is typically too computationally expensive since we use the full training set for each iteration.

- If $b = 1$, then we recover stochastic gradient descent, i.e., we train our model on a different example during each iteration. We lose all the speed related to vectorization, since we're dealing with single examples during each iteration.

- Choose $1 < b < n$ is typically always the best solution, since it deals with both of the aforementioned problems.

- Due to the nature of a computer's internal structure, it's typically better to choose a batch size $b$ for the form

  $$b = 2^p,$$

  for some $p \in \{6, 7, 8, 9, 10\}$ (usually $p < 10$).

- Choose a batch size $b$ that ensures your computer's CPU/GPU can hold a dataset of that size.

## 7.1 Weighted Averages

Suppose $x_t \in \mathbb{R}^m$ is some collection of data indexed by $t$ which we may consider a time-variable, that is, after each successive unit of time (say for example, each day), our collection adds a new data point. That is, the collection

$$\{x_t \in \mathbb{R}^m : 1 \leq t \leq T\}$$

has variable $T$.

Then if $X$ is the random vector associated to $x$, our usual mean $\mu$ is given by

$$\mu(T) := \mathbb{E}[X] = \frac{1}{T}\sum_{t=1}^{T} x_t.$$

Since our collection of data is growing and evolving over time, it's reasonable in many applications to have the most recent data points affect a model more than older data points. That is, we wish to impose a "weight" on more recent data points.

One way (and likely the most trivial) to achieve such a weighing is to have only the most recent $k$ examples affect our model. That is, for fixed $k \in \mathbb{N}$, and $t \geq k$, define the vector $\hat{x}_{t+1} \in \mathbb{R}^m$ by

$$\hat{x}_{t+1} = \frac{1}{k}\sum_{j=t-mk+1}^{t} x_j.$$

Then $\hat{x}_{t+1}$ represents the mean of the most recent $k$-examples. This may be interpreted as the "predicted-value" for $x_{t+1}$. This predictive model is known as a *simple moving average, or SMA*.

The simple moving average satisfies our weight requirement of focusing more on the most recent data, however, older data, though being less relevant, should still affect our model, but in a reduced form. The simple model does not satisfy this more refined requirement. Let's modify the simple model as follows: Fix $\beta_1 \in [0, 1)$ and we initialize a $V_0 = 0 \in \mathbb{R}^m$, and define recursively the vector $V_t \in \mathbb{R}^m$ given by

$$V_t = \beta_1 V_{t-1} + (1 - \beta_1)x_t.$$

We claim that $V_t$ can be interpreted as the next predicted value $\hat{x}_{t+1}$. Indeed, expanding our recursive definition

$$\begin{aligned}
V_t &= \beta_1 V_{t-1} + (1-\beta_1)x_t \\
&= \beta_1(\beta_1 V_{t-2} + (1-\beta_1)x_{t-1}) + (1-\beta_1)x_t \\
&= \beta_1^2 V_{t-2} + (1-\beta_1)(\beta_1 x_{t-1} + x_t) \\
&= \beta_1^2(\beta_1 V_{t-3} + (1-\beta_1)x_{t-2}) + (1-\beta_1)(\beta_1 x_{t-1} + x_t) \\
&= \beta_1^3 V_{t-3} + (1-\beta_1)(\beta_1^2 x_{t-2} + \beta_1 x_{t-1} + x_t) \\
&\vdots \\
&= \beta_t^t V_0 + (1-\beta_1)\sum_{j=0}^{t-1} \beta_1^j x_{t-j} \\
&= (1-\beta_1)\sum_{j=0}^{t-1} \beta_1^j x_{t-j}.
\end{aligned}$$

Moreover, if we define a probability distribution $\mathbb{P}$ as given by

$$\mathbb{P}(X = x_j) = (1-\beta_1)\beta_1^j,$$

then we immediately see that $V_t$ is the weighted-average over the last $t$-days, and hence may be interpreted as the predicted-value $\hat{x}_{t+1}$ as desired. Finally, since

$$1-\beta_1 = \frac{1}{\frac{1}{1-\beta_1}},$$

we may interpret $\frac{1}{1-\beta_1}$ as the size of the relevant sampling, i.e., $V_t$ is the average of $x$ over the previous $\frac{1}{1-\beta_1}$ days (assuming our time-units are measured in days). This predictive model is known as an *exponentially moving average, or EMA*.

**Remark 7.1.** *We note that since we initialize our EMA with $V_0 = 0$, that our predictive model is very bad for small $t$. This usually is irrelevant for many models, but if we need to correct for bias, we may make the modification of*

$$V_t = \frac{\beta_1 V_{t-1} + (1-\beta_1)x_t}{1-\beta_1^t}.$$

*Indeed, since $\beta_1 \in [0,1)$, we note that*

$$\begin{aligned}
\frac{1}{1-\beta_1} &= \sum_{j=0}^{\infty} \beta_1^j \\
&= \sum_{j=t}^{\infty} \beta_t^j + \sum_{j=0}^{t-1} \beta_1^j \\
&= \beta_1^t \sum_{j=0}^{\infty} \beta_1^j + \sum_{j=0}^{t-1} \beta_1^j \\
&= \frac{\beta_1^t}{1-\beta_1} + \sum_{j=0}^{t-1} \beta_1^j,
\end{aligned}$$

*and so*

$$\sum_{j=0}^{t-1} \beta_1^j = \frac{1-\beta_1^t}{1-\beta_1}.$$

*We then see that*

$$\begin{aligned}
V_t &= \frac{\beta_1 V_{t-1} + (1-\beta_1)x_t}{1-\beta_1^t} \\
&= \frac{(1-\beta_1)\sum_{j=0}^{t-1} \beta_1^j x_{t-j}}{1-\beta_1^t} \\
&= \frac{\sum_{j=0}^{t-1} \beta_1^j x_{t-j}}{\sum_{j=0}^{t-1} \beta_1^j},
\end{aligned}$$

*which is the explicit definition of a weighted-average.*

## 7.2 Gradient Descent with Momentum

Gradient descent has an issue with potentially plateauing during areas with a flat gradient, or bouncing around drastically before arriving at a minimum. One reason for this is that each iterative step only depends on the previous value of the gradient (or rather, the most recently updated parameter). The algorithm doesn't see larger trends, and so this leads to give our algorithm more history of the movements. We do this by using EMA.

We first recall our gradient descent algorithm:

1. We initialize $W^{\{0\}}$ and $b^{\{0\}}$.

2. For $0 \leq i <$ `num_iters`:

   a. Let $B = \left\lceil \frac{n}{b} \right\rceil$, and generate batches $\{\mathbb{X}^k\}$.

   b. For $1 \leq k \leq B$:

      i. Apply forward propagation on $\mathbb{X}^k$.

      ii. Compute the cost $\mathbb{J}$ on $\mathbb{X}^k$.

      iii. Apply backward propagation on $\mathbb{X}^k$ to obtain

      $$\frac{\partial \mathbb{J}}{\partial W}^{\{t\}}, \qquad \frac{\partial \mathbb{J}}{\partial b}^{\{t\}}.$$

      iv. We update parameters

      $$\begin{aligned} W^{\{t\}} &= W^{\{t-1\}} - \alpha \frac{\partial \mathbb{J}}{\partial W}^{\{t\}} \\ b^{\{t\}} &= b^{\{t-1\}} - \alpha \frac{\partial \mathbb{J}}{\partial b}^{\{t\}} \end{aligned}$$

Using this formulation of gradient descent, we insert EMA applied to the sequences of gradients depending on the iteration $t := i + k$. That is, we have the following algorithm:

1. Initialize our parameters $W^{\{0\}}$ and $b^{\{0\}}$. Initialize ${V_W}^{\{0\}} = {V_b}^{\{0\}} = 0$. Fix a momentum hyper-parameter $\beta_1 \in [0, 1)$.

2. For $0 \leq i <$ `num_iters`:

   a. Let $B = \left\lceil \frac{n}{b} \right\rceil$, and generate batches $\{\mathbb{X}^k\}$.

   b. For $1 \leq k \leq B$:

      i. Apply forward propagation on $\mathbb{X}^k$.

      ii. Compute the cost $\mathbb{J}$ on $\mathbb{X}^k$.

      iii. Apply backward propagation on $\mathbb{X}^k$ to obtain

      $$\frac{\partial \mathbb{J}}{\partial W}^{\{t\}}, \qquad \frac{\partial \mathbb{J}}{\partial b}^{\{t\}}.$$

      iv. Define

      $$\begin{aligned} {V_W}^{\{t\}} &= \beta_1 {V_W}^{\{t-1\}} + (1 - \beta_1) \frac{\partial \mathbb{J}}{\partial W}^{\{t\}} \\ {V_b}^{\{t\}} &= \beta_1 {V_b}^{\{t-1\}} + (1 - \beta_1) \frac{\partial \mathbb{J}}{\partial b}^{\{t\}} \end{aligned}$$

v. We update parameters

$$W^{\{t\}} = W^{\{t-1\}} - \alpha {V_W}^{\{t\}}$$
$$b^{\{t\}} = b^{\{t-1\}} - \alpha {V_b}^{\{t\}}$$

## 7.3 Root Mean Squared Propagation (RMSProp)

One of the main drawbacks to gradient descent with momentum is the uniformity of the modification regardless of the direction. That is, suppose our desired minimum is in the $\vec{b}$ direction, but the gradient $\partial_b \mathbb{J}$ is small while the gradient $\partial_W \mathbb{J}$ is large. As a result, our steps will oscillate wildly in the $\vec{w}$ direction, while moving very slowing in the $\vec{b}$ direction to our desired minimum. This as a whole can be very computationally slow, and is undesired.

The main idea for fixing these oscillatory issues is have a variable learning rate $\alpha$ which also depends on the direction. That is, if $\partial_W \mathbb{J}$ is large, and not in our desired direction of motion, we would like our update for $W$ to be small, and vice-versa if $\partial_b \mathbb{J}$ is small. Moreover, we wish to exaggerate the magnitudes of these vectors so we ensure our algorithm works efficiently. That is, we relate some vector $S$ via

$$S \sim \frac{\partial \mathbb{J}}{\partial W}^{2},$$

where we're taking that Hadamard-square (i.e., component-wise product with itself). Then we perform step via

$$W = W - \alpha \frac{1}{\sqrt{S}} \odot \frac{\partial \mathbb{J}}{\partial W},$$

where where taking the Hadamard-root. Note that this root is necessary for our update to make sense (consider the units involved in such an equation), but it does introduce the potential to divide by zero (which we'll fix by a small . Moreover, we would like use the history of gradients as in EMA to further our refinement of the descent algorithm. To this end, we have the following *RMSProp algorithm*:

1. Initialize our parameters $W^{\{0\}}$ and $b^{\{0\}}$. Initialize ${S_W}^{\{0\}} = {S_b}^{\{0\}} = 0$. Fix a momentum $\beta_2 \in [0, 1)$ and let $\epsilon > 0$ be sufficiently small ($\epsilon = 10^{-8}$ is a good starting point).

2. For $0 \leq i <$ `num_iter`:

a. Let $B = \left\lceil \frac{n}{b} \right\rceil$, and generate batches $\{\mathbb{X}^k\}$

b. For $1 \leq k \leq B$:

   i. Apply forward propagation on $\mathbb{X}^k$.

   ii. Compute the cost $\mathbb{J}$ on $\mathbb{X}^k$.

   iii. Apply backward propagation on $\mathbb{X}^k$ to obtain

$$\frac{\partial \mathbb{J}}{\partial W}^{\{t\}}, \qquad \frac{\partial \mathbb{J}}{\partial b}^{\{t\}}.$$

   iv. Define

$$S_W{}^{\{t\}} = \beta_2 S_W{}^{\{t-1\}} + (1 - \beta_2)\left(\frac{\partial \mathbb{J}}{\partial W}^{\{t\}}\right)^2$$

$$S_b{}^{\{t\}} = \beta_2 S_b{}^{\{t-1\}} + (1 - \beta_2)\left(\frac{\partial \mathbb{J}}{\partial b}^{\{t\}}\right)^2$$

   v. Update parameters via

$$W^{\{t\}} = W^{\{t-1\}} - \alpha \frac{\frac{\partial \mathbb{J}}{\partial W}^{\{t\}}}{\sqrt{S_W{}^{\{t\}}} + \epsilon}$$

$$b^{\{t\}} = b^{\{t-1\}} - \alpha \frac{\frac{\partial \mathbb{J}}{\partial b}^{\{t\}}}{\sqrt{S_b{}^{\{t\}}} + \epsilon}$$

## 7.4 Adaptive Moment Estimation: The Adam Algorithm

We first note that with the momentum algorithm utilizing the EMA as it does, that it is an algorithm of the first moment (i.e., the mean of the gradients). Similarly, with RMSProp utilizing the square of the gradient as it does, we say it is an algorithm of the second moment (i.e., the uncentered variance of the gradients). Our goal it utilize both gradient descent with momentum and RMSProp simultaneously to optimize our parameters. This combination of algorithms is called the *Adam algorithm* and is implemented as follows:

1. Initialize our parameters $W^{\{0\}}$ and $b^{\{0\}}$. Initialize $V_W{}^{\{0\}} = V_b{}^{\{0\}} = 0$ and $S_W{}^{\{0\}} = S_b{}^{\{0\}} = 0$. Fix our constants of momenta $\beta_1, \beta_2 \in [0, 1)$ and let $\epsilon > 0$ be sufficiently small.

2. For $0 \leq i <$ `num_iters`:

   a. Let $B = \left\lceil \frac{n}{b} \right\rceil$, and generate batches $\{\mathbb{X}^k\}$

   b. For $1 \leq k \leq B$:

      i. Apply forward propagation on $\mathbb{X}^k$.

      ii. Compute the cost $\mathbb{J}$ on $\mathbb{X}^k$.

      iii. Apply backward propagation on $\mathbb{X}^k$ to obtain

$$\frac{\partial \mathbb{J}}{\partial W}^{\{t\}}, \qquad \frac{\partial \mathbb{J}}{\partial b}^{\{t\}}.$$

      iv. Define

$$V_W{}^{\{t\}} = \beta_1 V_W{}^{\{t-1\}} + (1-\beta_1)\frac{\partial \mathbb{J}}{\partial W}^{\{t\}},$$
$$V_b{}^{\{t\}} = \beta_1 V_b{}^{\{t-1\}} + (1-\beta_1)\frac{\partial \mathbb{J}}{\partial b}^{\{t\}},$$

      and define

$$S_W{}^{\{t\}} = \beta_2 S_W{}^{\{t-1\}} + (1-\beta_2)\left(\frac{\partial \mathbb{J}}{\partial W}^{\{t\}}\right)^2,$$
$$S_b{}^{\{t\}} = \beta_2 S_b{}^{\{t-1\}} + (1-\beta_2)\left(\frac{\partial \mathbb{J}}{\partial b}^{\{t\}}\right)^2.$$

      v. Utilize bias correction via:

$$\hat{V_W}^{\{t\}} = \frac{V_W{}^{\{t\}}}{1-\beta_1^t}$$
$$\hat{V_b}^{\{t\}} = \frac{V_b{}^{\{t\}}}{1-\beta_1^t}$$
$$\hat{S_W}^{\{t\}} = \frac{S_W{}^{\{t\}}}{1-\beta_2^t}$$
$$\hat{S_b}^{\{t\}} = \frac{S_b{}^{\{t\}}}{1-\beta_2^t}$$

vi. Update the parameters:

$$W^{\{t\}} = W^{\{t-1\}} - \alpha \frac{\hat{V_W}^{\{t\}}}{\sqrt{\hat{S_W}^{\{t\}}} + \epsilon}$$

$$b^{\{t\}} = b^{\{t-1\}} - \alpha \frac{\hat{V_b}^{\{t\}}}{\sqrt{\hat{S_b}^{\{t\}}} + \epsilon}$$

We note that though we may still need to tune the hyper-parameter $\alpha$, the hyper-parameters $\beta_1, \beta_2$ and $\epsilon$ typically work quite well with default values of

$$\beta_1 = 0.9, \qquad \beta_2 = 0.999, \qquad \epsilon = 10^{-8}.$$

## 7.5 Learning Rate Decay

Finally, one further method we may utilize in our optimization problem, is the idea of slowly reducing our learning rate $\alpha$. That is, if $i$ is our epoch iteration, and $\eta > 0$ is a fixed decay rate, we can define new learning rates in many ways. That is, for $\alpha = \alpha(i)$ we can define

- $$\alpha(i) = \frac{1}{1 + \eta i}\alpha_0,$$
- $$\alpha(i) = \alpha_0 \eta^i,$$
- $$\alpha(i) = \frac{\eta}{\sqrt{i}}\alpha_0.$$

One could also implement a "manual decay", but this should only be used under ideal circumstances.

## 7.6 Python Implementation

```
import copy

import numpy as np
from sklearn.utils import shuffle

```

```python
import mlLib.utils as utils

def get_batches(x, y, b):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (m, n)
    y : array_like
        y.shape = (k, n)
    b : int

    Returns
    -------
    batches : List[Dict]
        batches[i][’x’] : array_like
            x.shape = (m, b) # except last batch
            y.shape = (k, b) # except last batch

    """
    m, n = x.shape
    ## Shuffle the data
    x, y = shuffle(x.T, y.T) # Only shuffles rows, so transpose is needed
    x = x.T
    y = y.T

    B = int(np.ceil(n / b))
    batches = []
    for i in range(B):
        x_temp = x[:,(b * i):(b * (i + 1))]
        y_temp = y[:,(b * i):(b * (i + 1))]
        batches.append({’x’ : x_temp, ’y’ : y_temp})
    # Slicing automatically ends at the end of
    # the list if the stop is outside the index
    return batches

def initialize_momenta(layers):
    """
    Parameters
    ----------
    layers : List[int]
        layers[l] = # nodes in layer l
    Returns
    -------
    v : Dict[Dict[array_like]]
    s : Dict[Dict[array_like]]
    """
    vw = {}
    vb = {}
    sw = {}
    sb = {}
    for l in range(1, len(layers)):
        vw[l] = np.zeros((layers[l], layers[l - 1]))
        sw[l] = np.zeros((layers[l], layers[l - 1]))
        vb[l] = np.zeros((layers[l], 1))
        sb[l] = np.zeros((layers[l], 1))

    v = {’w’ : vw, ’b’ : vb}
    s = {’w’ : sw, ’b’ : sb}

    return v, s

def learning_rate_decay(epoch, learning_rate=0.01, decay_rate=0.0):
    """
    Parameters
    ----------
    eposh : int
    learning_rate : float
        Default: 0.01
    decay_rate : float
        Default: 0.0 - Returns a constant learning_rate

    Returns
    -------
    learning_rate : float
    """
    learning_rate = (1 / (1 + epoch * decay_rate)) * learning_rate
    return learning_rate

def corrected_momentum(v, grads, update_iter, beta1=0.0):
    """
    Parameters
    ----------
    v : Dict[Dict[array_like]]
        v[’w’][l].shape = w[l].shape
        v[’b’][l].shape = b[l].shape
    grads : Dict[Dict]
        grads[’w’][l] : array_like
            dw[l].shape = w[l].shape
        grads[’b’][l] : array_like
            db[l].shape = b[l].shape
    update_iter : int
    beta1 : float
        Default: 0.0 - Returns grads
        Usual: 0.9

    Returns
    -------
    v : Dict[Dict[array_like]]
        v[’w’][l].shape = dw[l].shape
        v[’b’][l].shape = db[l].shape
    """
    ## Retrieve velocities and gradients
    vw = v[’w’]
    vb = v[’b’]
    dw = grads[’w’]
    db = grads[’b’]
    L = len(dw)

    for l in range(1, L + 1):
        vw[l] = beta1 * vw[l] + (1 - beta1) * dw[l]
        vw[l] /= (1 - beta1 ** update_iter)
        assert(vw[l].shape == dw[l].shape)
        vb[l] = beta1 * vb[l] + (1 - beta1) * db[l]
        vb[l] /= (1 - beta1 ** update_iter)
        assert(vb[l].shape == db[l].shape)

    v = {’w’ : vw, ’b’ : vb}
    return v

def corrected_rmsprop(s, grads, update_iter, beta2=0.999):
    """
    Parameters
    ----------
    s : Dict[Dict[array_like]]
        s[’w’][l].shape = w[l].shape
        s[’b’][l].shape = b[l].shape
    grads : Dict[Dict]
        grads[’w’][l] : array_like
            dw[l].shape = w[l].shape
        grads[’b’][l] : array_like
            db[l].shape = b[l].shape
    update_iter : int
    beta2 : float
        Default: 0.999

    Returns
    -------
    s : Dict[Dict[array_like]]
        s[’w’][l].shape = w[l].shape
        s[’b’][l].shape = b[l].shape
    """
    ## Retrieve accelerations and gradients
    sw = s[’w’]
    sb = s[’b’]
    dw = grads[’w’]
    db = grads[’b’]
    L = len(dw)

    for l in range(1, L + 1):
        sw[l] = beta2 * sw[l] + (1 - beta2) * (dw[l] * dw[l])
        sw[l] /= (1 - beta2 ** update_iter)
        assert(sw[l].shape == dw[l].shape)
        sb[l] = beta2 * sb[l] + (1 - beta2) * (db[l] * db[l])
        sb[l] /= (1 - beta2 ** update_iter)
        assert(sb[l].shape == db[l].shape)

    s = {’w’ : sw, ’b’ : sb}
    return s


def update_parameters_adam(params, grads, epoch, batch_iter, v, s, momenta=[1e-8, 0.
    """
    Parameters
    ----------
    params : Dict[Dict]
        params[’w’][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            b[l].shape = (layers[l], 1)
    grads : Dict[Dict]
        grads[’dw’][l] : array_like
            dw[l].shape = w[l].shape
        grads[’db’][l] : array_like
            db[l].shape = b[l].shape
    epoch : int
    batch_iter : int
    learning_rate : float
        Default: 0.01
    momenta : List[float]
        momenta[0] = epsilon
            Default: 10^{-8}
        momenta[1] = beta_1
            Default: 0.9
        momenta[2] = beta_2
            Default: 0.999

    Returns
    -------
    params : Dict[Dict]
        params[’w’][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            b[l].shape = (layers[l], 1)
    ”””
    update_iter = epoch + batch_iter
    ## Retrieve parameters
    w = copy.deepcopy(params[’w’])
    b = copy.deepcopy(params[’b’])
    L = len(w)

    ## Update velocites and accelerations
    v = corrected_momentum(v, grads, update_iter, momenta[1])
    vw = v[’w’]
    vb = v[’b’]
    s = corrected_rmsprop(s, grads, update_iter, momenta[2])
    sw = s[’w’]
    sb = s[’b’]

    ## Update learning rate
    learning_rate = learning_rate_decay(epoch, alpha0, decay_rate)

    ## Perform update
    for l in range(1, L + 1):
        w[l] = w[l] - learning_rate * vw[l] / (np.sqrt(sw[l]) + momenta[0])
        b[l] = b[l] - learning_rate * vb[l] / (np.sqrt(sb[l]) + momenta[0])

    params = {’w’ : w, ’b’ : b}
    return params

def model(x, y,
        hidden_layer_sizes,
        activators,
        batch_size,
        lambda_=0.0,
        num_iters=10000,
        print_cost=False):
    ”””
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[-1], n)
    hidden_layer_sizes : List[int]
```

```
        The number nodes layer l = hidden_layer_sizes[l-1]
    activators : List[str]
        activators[l] = activation function of layer l+1
    batch_size : int
    lambda_ : float
        The regularization parameter
        Default: 0.0
    num_iters : int
        Number of iterations with which our model performs gradient descent
        Default: 10000
    print_cost : Boolean
        If True, print the cost every 1000 iterations
        Default: False

    Returns
    -------
    params : Dict[Dict]
        params[’w’][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            b[l].shape = (layers[l], 1)
    cost : float
        The final cost value for the optimized parameters returned
    """
    n, layers = utils.dim_retrieval(x, y, hidden_layer_sizes)
    params = utils.initialize_parameters_random(layers)
    v, s = initialize_momenta(layers)


    ## main descent loop
    for i in range(num_iters):
        batches = get_batches(x, y, batch_size)
        ## batch loop
        batch_iter = 1
        cost = 0
        for batch in batches:
            x = batch[’x’]
            y = batch[’y’]
            cache = utils.forward_propagation(x, params, activators)
            cost += utils.compute_cost(y, params, cache)
            grads = utils.backward_propagation(x, y, params, cache, activators)
            params =  update_parameters_adam(params,
                                       grads,
                                       i,
                                       batch_iter,
                                       v,
                                       s,
                                       momenta=[1e-8, 0.9, 0.999],
                                       learning_rate=0.01,
                                       decay_rate = 0.0)
            batch_iter += 1

        if print_cost and i % 1000 == 0:
            print(f'Cost␣after␣iteration␣{i}:␣{cost}')

    return params, cost
```

# 8 Tuning Hyper-Parameters

Suppose that we have the dataset $\mathbb{D}$ with the usual partition of

$$\mathbb{D} = \mathbb{X} \cup \mathcal{D} \cup \mathcal{T}.$$

Furthermore, suppose we impose a neural network architecture which has a collection of hyper-parameters (relabeled as):

$$\eta_1, \eta_2, ...\eta_K.$$

The naive method of hyper-parameter tuning would instinctively be something of the form: Let $[d_i, d_i + k_i\Delta_i]$ denote an interval for which we require

$$\eta_i \in [d_i, d_i + k_i\Delta_i],$$

with an even-partition of

$$d_i < d_i + \Delta_i < d_i + 2\Delta_i < \cdots < d_i + k_i\Delta_i,$$

of length $\Delta_i$. This collection forms a "grid" in $\mathbb{R}^K$ for which each point of the grid gives us a full collection of hyper-parameters which we can then use to train our model. However, if certain hyper-parameters do not affect our model's accuracy very much, we've added at least a full dimension of validation which is not needed. A more randomized approach would be best to determine such a hyper-parameter characterization must faster. Thus a random collection of points $H_i$ for which we constrain $\eta_i \in H_i$.

How should we implement this set $H_i$? Suppose for example, we wish to find

$$\eta_i \in [0.0001, 1],$$

but the majority of the random points will likely be in $[0.1, 1]$. Suppose we partition the interval

$$\begin{aligned}[0.0001, 1] &= 0.0001 < 0.001 < 0.01 < 0.1 < 1 \\ &= 10^{-4} < 10^{-3} < 10^{-2} < 10^{-1} < 10^{0}.\end{aligned}$$

This suggests we obtain a distribution of points using a logarithmic (in base 10) scale. Indeed, let

$$p \in [0, 1],$$

be a random point. Then letting $r = -4p \in [-4, 0]$, we obtain another random point, and let

$$H_i = \{10^{-4p} : p \in \text{rand}([0, 1])\},$$

for some prescribed set-cardinality. This allows us to choose more appropriately scaled-options for our hyper-parameters.

**Remark 8.1.** *Suppose we're using exponentially moving averages and have a hyper-parameter $\beta_1 \in [0, 1)$. If we do not use a log-scale, then the sensitivity of our model with respect to $\beta_1$ when $\beta_1 \approx 1$ is very strong. Indeed, we recall that when $\beta_1 = 0.999$, this corresponds to averaging over the previous* 1000 *days. And it we change $\beta_1$ slightly to*

$$\beta_1 = 0.9995,$$

*then we've changed the interpretation of our model to the previous* 2000 *days. A subtle change for $\beta_1$, but a drastic change to our model. The log-scale fixes this issue immediately.*

We finally note that our hyper-parameters can become *stale* over time. That is, suppose we've trained a neural network, and tuned the hyper-parameters to allow an acceptable accuracy for our model. As the model refines over time, with more data being inserted to train on, it's import to re-test our hyper-parameters to make sure our model hasn't opened up to a better choice of one (or some or all) of the hyper-parameters we've previously tuned.

## 8.1 Python Implementation

```
def hyperparameter_scale(k, p):
    """
    Parameters
    ----------
    k : int
        The number random points to generate
    p : int
        The smallest magnitude for our log-scale

    Returns
    -------
    hypers : List[float]
        The list of hyper-parameters with which to tune
    """
    hypers = []
    for _ in range(k):
        r = p * np.random.rand()
        hypers.append(10 ** r)
    return hypers
```

# 9 Batch Normalization

See [1].

We recall feature-normalization: Suppose $x \in \mathbb{R}^{m\times n}$ is some training data, and let

$$\mu = \mathbb{E}[X], \qquad \sigma^2 = \mathbb{E}[(X-\mu)^2],$$

denote the mean and variance of the random-vector representation $X$ of $x$, respectively. Then we consider the map

$$x_j \mapsto \frac{x_j - \mu}{\sigma} =: \hat{x}_j,$$

to be the *normalization* of $x_j$.

This definition is so “vanilla”, that it should be clear that this can be easily applied to each hidden-layer (we shall not use it on the output layer) of a neural network as well. However, we first note that there is an ambiguous choice amongst the implementation, namely, do we normalize $z^{[\ell]}$ or $a^{[\ell]}$, i.e., does normalization occur before or after we compute the activation unit. It seems more common to apply normalization to $z^{[\ell]}$, so that is what we do here without further mention of this choice.

Let $\gamma, \beta \in \mathbb{R}^m$, if we consider the map

$$\hat{x}_j \mapsto \gamma \odot \hat{x}_j + \beta := \tilde{x}_j,$$

we can see fairly trivially that we can recover $x_j$ (thus allowing for identity activation units), indeed, let $\gamma = \sigma$ and $\beta = \mu$, and hence

$$\begin{aligned}\tilde{x}_j &= \gamma \odot \hat{x}_j + \beta \\ &= \gamma \odot \frac{x_j - \mu}{\sigma} + \beta \\ &= x_j - \mu_\beta \\ &= x_j\end{aligned}$$

as desired. Moreover, we see that we can actually control what mean and variance we wish to impose on our input-vectors $x$. Indeed, let $\hat{x}$ denote the normalized $x$, and consider

$$\begin{aligned}\mathbb{E}[\gamma \odot \hat{X} + \beta] &= \frac{1}{n}\sum_{j=1}^{n}(\gamma \odot \hat{x}_j + \beta) \\ &= \gamma \odot \mathbb{E}[\hat{X}] + \beta \\ &= 0 + \beta \\ &= \beta,\end{aligned}$$

and so the new mean would be given by $\beta$. Similarly,

$$\begin{aligned}\mathbb{E}[(\gamma \odot \hat{X} + \beta - \beta)^2] &= \frac{1}{n}\sum_{j=1}^{n}(\gamma \odot \hat{x}_j)^2 \\ &= \frac{1}{n}\sum_{j=1}^{n}(\gamma^2 \odot \hat{x}_j^2) \\ &= \gamma^2 \odot \mathbb{E}[(\hat{X} - 0)^2] \\ &= \gamma^2 \odot 1 \\ &= \gamma^2\end{aligned}$$

and so we see the new variance would be given by $\gamma^2$. Thus, we see that by composition, the act of normalization can be characterized by the new parameters $\gamma$ and $\beta$, and is mathematically-superfluous to consider both, but for computational considerations and algorithmic stability it shall be beneficial to keep both. That is, suppose we're training on some batch $\mathbb{X}^k$ and focused on layer-$\ell$, with parameters $\gamma^{[\ell]}, \beta^{[\ell]} \in \mathbb{R}^{m_\ell}$ and some $\epsilon > 0$, arbitrarily small and prescribed for numerical stability, we define the *batch-normalization* map $BN_{\gamma^{[\ell]},\beta^{[\ell]}} : \mathbb{R}^{m_\ell} \to \mathbb{R}^{m_\ell}$ given by the compositional-map

$$\begin{aligned}z^{[\ell]} &\mapsto \frac{1}{|\mathbb{X}^k|}\sum_{x\in\mathbb{X}^k} z^{[\ell]} =: \mu^{[\ell]}; \\ (z^{[\ell]}, \mu^{[\ell]}) &\mapsto \frac{1}{|\mathbb{X}^k|}\sum_{x\in\mathbb{X}^k} (z^{[\ell]} - \mu^{[\ell]})^2 =: \sigma^{[\ell]2}; \\ (z^{[\ell]}, \mu^{[\ell]}, \sigma^{[\ell]}, \epsilon) &\mapsto \frac{z^{[\ell]} - \mu^{[\ell]}}{\sqrt{\sigma^{[\ell]2} + \epsilon}} =: \hat{z}^{[\ell]}; \\ (\hat{z}^{[\ell]}, \gamma^{[\ell]}, \beta^{[\ell]}) &\mapsto \gamma^{[\ell]} \odot \hat{z}^{[\ell]} + \beta^{[\ell]} =: \tilde{z}^{[\ell]}.\end{aligned}$$

Suppose we have an $L$-layer neural network, each layer with $m_\ell$ nodes, and we focus on the $\ell$-th layer specifically to expand:

$$\cdots \xrightarrow{\varphi^{[\ell]}} \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]m_\ell} \end{bmatrix} \xrightarrow{BN_{\gamma^{[\ell]},\beta^{[\ell]}}} \begin{bmatrix} \tilde{z}^{[\ell]1} \\ \vdots \\ \tilde{z}^{[\ell]}{}_{m_\ell} \end{bmatrix} \xrightarrow{g^{[\ell]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]m_\ell} \end{bmatrix}}_{\text{Layer } \ell} \xrightarrow{\varphi^{[\ell+1]}} \cdots$$

The procedure for forward propagation should be immediately obvious from the closer look at layer-$\ell$. However, we notice that

$$\begin{aligned} a^{[\ell-1]} &\mapsto \gamma^{[\ell]} \odot \frac{W^{[\ell]}a^{[\ell-1]} + b^{[\ell]} - \mu^{[\ell]}}{\sqrt{\sigma^{[\ell]2} + \epsilon}} + \beta^{[\ell]} \\ &= \frac{\gamma^{[\ell]}}{\sqrt{\sigma^{[\ell]2} + \epsilon}}(W^{[\ell]}a^{[\ell-1]} - \mu^{[\ell]}) + \beta^{[\ell]}, \end{aligned}$$

after absorbing the $b^{[\ell]}$ into the parameter $\beta^{[\ell]}$. That is, we have 3 trainable parameters given by $W^{[\ell]} \in \mathbb{R}^{m_\ell \times m_{\ell-1}}$, $\gamma^{[\ell]}, \beta^{[\ell]} \in \mathbb{R}^{m_\ell}$.

## 9.1 Backward Propagation

We now show how batch normalization affects the backward propagation algorithm. For illustrative purposes, we assume a 2-layer neural network with arbitrary activation functions and generic loss function. We recall the setup (without bias $b^{[\ell]}$) used in Section 2.1

$$\underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{m_0} \end{bmatrix}}_{\text{Layer } 0} \xrightarrow{\Phi^{[1]}} \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]m_1} \end{bmatrix} \xrightarrow{BN_{\gamma,\beta}} \begin{bmatrix} \tilde{z}^{[1]1} \\ \vdots \\ \tilde{z}^{[\ell]}{}_{m_\ell} \end{bmatrix} \xrightarrow{g^{[1]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]m_1} \end{bmatrix}}_{\text{Layer } 1} \xrightarrow{\Phi^{[2]}} \cdots$$

$$\cdots \xrightarrow{\Phi^{[2]}} \underbrace{\begin{bmatrix} z^{[2]1} \\ \vdots \\ z^{[2]m_2} \end{bmatrix} \xrightarrow{g^{[2]}} \begin{bmatrix} a^{[2]1} \\ \vdots \\ a^{[2]m_2} \end{bmatrix}}_{\text{Layer } 2} \xrightarrow{=} \begin{bmatrix} \hat{y}^1 \\ \vdots \\ \hat{y}^{m_2} \end{bmatrix},$$

where

$$\Phi^{[1]} : \mathbb{R}^{m_1 \times m_0} \times \mathbb{R}^{m_0} \to \mathbb{R}^{m_1}, \qquad \Phi^{[1]}(A, x) = Ax;$$

and

$$\Phi^{[2]} : \mathbb{R}^{m_2 \times m_1} \times \mathbb{R}^{m_2} \times \mathbb{R}^{m_1} \to \mathbb{R}^{m_2}, \qquad \Phi^{[2]}(A, b, x) = Ax + b.$$

Define the compositional function

$$G : \mathbb{R}^{m_2 \times m_1} \times \mathbb{R}^{m_2} \times \mathbb{R}^{m_1} \times \mathbb{R}^{m_1} \times \mathbb{R}^{m_1 \times m_0} \times \mathbb{R}^{m_0} \to \mathbb{R},$$

given by

$$G(B, b, \gamma, \beta, A, x) = \mathbb{L}_y \circ g^{[2]} \circ \Phi^{[2]}(B, b, g^{[1]} \circ BN_{\gamma,\beta}(\Phi^{[1]}(A, x))).$$

Since we don't use batch normalization on the output layer, the bias term still exists.

This leads to compute some auxiliary differentials before continuing further.

**Lemma 9.1.** *For $N \in \mathbb{N}$, we define the expectation function $\mathbb{E} : \mathbb{R}^N \to \mathbb{R}$ given by*

$$\mathbb{E}[(x_1, ..., x_N)] = \frac{1}{N} \sum_{j=1}^{N} x_j.$$

*Let $z = \{z_1, ..., z_N\} \subset \mathbb{R}$ be fixed, and define the mean*

$$\mu := \mathbb{E}[z] = \frac{1}{N} \sum_{j=1}^{N} z_j.$$

*Then as a differential, we have that $d\mathbb{E}_z : T_z\mathbb{R}^N \to T_\mu\mathbb{R}$ given by*

$$d\mathbb{E}_z = \frac{1}{N} \sum_{j=1}^{N} dx_j|_{x=z}, \qquad d\mathbb{E}_z(v) = \frac{1}{N} \sum_{j=1}^{N} v^j.$$

*Moreover, for $\alpha = 1, ..., N$, let $\iota_{z_\alpha} : \mathbb{R} \to \mathbb{R}^N$ denote the inclusion*

$$\iota_{z_\alpha}(x) = (z_1, ..., z_{\alpha-1}, x, z_{\alpha+1}, ..., z_N).$$

*Then the differentials*

$$d_\alpha \mathbb{E}_{z_\alpha} := d(\mathbb{E} \circ \iota_{z_\alpha})_{z_\alpha} : T_{z_\alpha}\mathbb{R} \to T_\mu\mathbb{R},$$

*are given by*

$$\begin{aligned} d_\alpha \mathbb{E}_{z_\alpha} &= d(\mathbb{E} \circ \iota_{z_\alpha})_{z_\alpha} \\ &= d\mathbb{E}_z \cdot d(\iota_{z_\alpha})_{z_\alpha} \\ &= \frac{1}{N} dx_{z_\alpha}. \end{aligned}$$

*Similarly, we define the variance function $\mathbb{V} : \mathbb{R}^N \to \mathbb{R}$ given by*

$$\mathbb{V}[(x_1, ..., x_N)] = \frac{1}{N}\sum_{j=1}^{N}(x_j - \mathbb{E}[(x_1, ..., x_N)])^2.$$

*For fixed $z$, define the variance*

$$\sigma^2 = \mathbb{V}[z].$$

*Then as a differential, we have that $d\mathbb{V}_z : T_z\mathbb{R}^N \to T_{\sigma^2}\mathbb{R}$ given by*

$$d\mathbb{V}_z = \frac{2}{N}\sum_{j=1}^{N}(z_j - \mu)\, dx^j\big|_{x=z}, \qquad d\mathbb{V}_z(v) = \frac{2}{N}\sum_{j=1}^{N}(z_j - \mu)v^j.$$

*Moreover, for $\alpha = 1, ..., N$, the differentials*

$$d_\alpha\mathbb{V}_{z_\alpha} := d(\mathbb{V} \circ \iota_{z_\alpha})_{z_\alpha} : T_{z_\alpha}\mathbb{R} \to T_{\sigma^2}\mathbb{R}$$

*are given by*

$$\begin{aligned} d_\alpha\mathbb{V}_{z_\alpha} &= d(\mathbb{V} \circ \iota_{z_\alpha})_{z_\alpha} \\ &= d\mathbb{V}_z \cdot d(\iota_{z_\alpha})_{z_\alpha} \\ &= \frac{2}{N}(z_\alpha - \mu)dx_{z_\alpha} \end{aligned}$$

**Proof:** Immediate from direct calculation. □

**Corollary 9.2.** *For $\alpha = 1, ..., N$, let $\mathcal{N}_\alpha : \mathbb{R}^{m\times N} \to \mathbb{R}^m$ denote the $\alpha$-th component of the vector-valued, normalization transformation. That is,*

$$\hat{x}_\alpha = \mathcal{N}_\alpha(x_1, ..., x_N),$$

*with*

$$\hat{x}^i_\alpha = \frac{\pi_\alpha(x^i) - \mathbb{E}[x^i]}{(\mathbb{V}[x^i] + \epsilon)^{\frac{1}{2}}},$$

*where $\pi_\alpha : \mathbb{R}^N \to \mathbb{R}$ is the projection onto the $\alpha$-th coordinate*

$$\pi_\alpha(x_1, ..., x_N) = x_\alpha.$$

*Fix $z_1, ..., z_N \in \mathbb{R}^m$, let $\mu = \mathbb{E}[z] \in \mathbb{R}^m$ denote vector-mean and let $\sigma^2 = \mathbb{V}[z] \in \mathbb{R}^m$ denote the component-wise, vector-variation (i.e., $(\sigma^2)^i = \mathbb{V}[z^i]$). Then the differentials*

$$d_\alpha(\mathcal{N}_\alpha)_{z_\alpha} := d(\mathcal{N}_\alpha \circ \iota_{z_\alpha})_{z_\alpha} : T_{z_\alpha}\mathbb{R}^m \to T_{\hat{z}_\alpha}\mathbb{R}^m$$

*are given by the diagonal matrices*

$$d_\alpha(\mathcal{N}_\alpha)_{z_\alpha} = \left( \frac{1 - \frac{1}{N}}{\sqrt{(\sigma^2)^i + \epsilon}} - \frac{1}{N} \frac{(z^i_\alpha - \mu^i)^2}{((\sigma^2)^i + \epsilon)^{\frac{3}{2}}} \right) \delta^i_j.$$

**Proof:** We compute directly after noting that

$$d_\alpha(\mathcal{N}_\alpha)_{z_\alpha} = \begin{bmatrix} d_\alpha(\hat{x}^1_\alpha)_{z^1_\alpha} & \cdots & 0 \\ 0 & \ddots & 0 \\ 0 & \cdots & d_\alpha(\hat{x}^m_\alpha)_{z^m_\alpha} \end{bmatrix}$$

To this end, fix $1 \leq i \leq m$ and we compute

$$\begin{aligned} d_\alpha(\hat{x}^i_\alpha)_{z^i_\alpha} &= d_\alpha(\mathcal{N}^i_\alpha)_{z^i_\alpha} \\ &= \frac{d_\alpha(\pi_\alpha)_{z^i_\alpha} - d_\alpha \mathbb{E}_{z^i_\alpha}}{\sqrt{(\sigma^2)^i + \epsilon}} - \frac{z^i_\alpha - \mu^i}{2((\sigma^2)^i + \epsilon)^{\frac{3}{2}}} d_\alpha \mathbb{V}_{z^i_\alpha} \\ &= \frac{dx_{z^i_\alpha} - \frac{1}{N} dx_{z^i_\alpha}}{\sqrt{(\sigma^2)^i + \epsilon}} - \frac{z^i_\alpha - \mu^i}{2((\sigma^2)^i + \epsilon)^{\frac{3}{2}}} \left( \frac{2}{N}(z^i_\alpha - \mu^i) dx_{z^i_\alpha} \right) \\ &= \left( \frac{1 - \frac{1}{N}}{\sqrt{(\sigma^2)^i + \epsilon}} - \frac{(z^i_\alpha - \mu^i)^2}{N((\sigma^2)^i + \epsilon)^{\frac{3}{2}}} \right) dz^i_\alpha, \end{aligned}$$

as desired. □

**Proposition 9.3.** *Let $\mathcal{N} : \mathbb{R}^{m \times N} \to \mathbb{R}^{m \times N}$ denote the usual normalization transformation with $\hat{x}_\alpha = \mathcal{N}_\alpha(x)$. Let $BN : \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^{m \times N} \to \mathbb{R}^{m \times N}$ denote the batch normalization transformation $[x_j] \mapsto [\tilde{x}_j]$, i.e.,*

$$\tilde{x}^i_j = \gamma^i \hat{x}^i_j + \beta^i,$$

*where $x^i \in \mathbb{R}^N$. Moreover, given $\gamma, \beta \in \mathbb{R}^m$, for $\alpha \in \{1, ..., N\}$, let*

$$BN^{\gamma,\beta}_\alpha : \mathbb{R}^{m \times N} \to \mathbb{R}^m$$

*denote*

$$BN^{\gamma,\beta}_\alpha(x) = \gamma \odot \mathcal{N}_\alpha(x) + \beta.$$

*Fix $z_1, ..., z_N, \in \mathbb{R}^m$, and let*

$$\hat{z}_\alpha = \mathcal{N}_\alpha(z_1, ..., z_N) \in \mathbb{R}^m, \qquad \mu^i = \mathbb{E}[z^i] \in \mathbb{R}, \qquad (\sigma^2)^i = \mathbb{V}[z^i] \in \mathbb{R}.$$

*For $\alpha \in \{1, ..., N\}$, $z \in \mathbb{R}^{m \times N}$ and for $\gamma, \beta \in \mathbb{R}^m$, we have the differentials:*

- $d(BN_\alpha^{\beta,z})_\gamma : T_\gamma \mathbb{R}^m \to T_{\tilde{z}} \mathbb{R}^m$, *is given by*

$$d(BN_\alpha^{\beta,z})_\gamma(v) = \hat{z}_\alpha \odot v, \qquad \frac{\partial \tilde{z}^i_\alpha}{\partial \gamma^j} = \hat{z}^i_\alpha \delta^i_j.$$

- $d(BN_\alpha^{\gamma,z})_\beta : T_\beta \mathbb{R}^m \to T_{\tilde{z}} \mathbb{R}^m$ *is given by*

$$d(BN_\alpha^{\gamma,z})_\beta(v) = v, \qquad \frac{\partial \tilde{z}^i_\alpha}{\partial \beta^j} = \delta^i_j.$$

- $d(BN_\alpha^{\gamma,\beta})_{\hat{z}_\alpha} : T_{\hat{z}_\alpha} \mathbb{R}^m \to T_{\tilde{z}} \mathbb{R}^m$ *is given by*

$$d(BN_\alpha^{\gamma,\beta})_{\hat{z}_\alpha}(v) = \gamma \odot v, \qquad \frac{\partial \tilde{z}^i_\alpha}{\partial \hat{z}^j_\alpha} = \gamma^i \delta^i_j.$$

- $d_\alpha(BN_\alpha^{\gamma,\beta})_{z_\alpha} := d(BN_\alpha^{\gamma,\beta} \circ \iota_{z_\alpha})_{z_\alpha} : T_{z_\alpha} \mathbb{R}^m \to T_{\tilde{z}_\alpha} \mathbb{R}^m$ *is given by*

$$d_\alpha(BN_\alpha^{\gamma,\beta})_{z_\alpha} = (\gamma \odot) d_\alpha(\mathcal{N}_\alpha)_{z_\alpha},$$

$$\frac{\partial \tilde{z}^i_\alpha}{\partial z^j_\alpha} = \gamma^i \left( \frac{1 - \frac{1}{N}}{\sqrt{(\sigma^2)^i + \epsilon}} - \frac{(z^i_\alpha - \mu^i)^2}{N((\sigma^2)^i + \epsilon)^{\frac{3}{2}}} \right) \delta^i_j$$

**Proof:** Follows immediately from the previous Corollary. □

We now return to considering the compositional function

$$G : \mathbb{R}^{m_2 \times m_1} \times \mathbb{R}^{m_2} \times \mathbb{R}^{m_1} \times \mathbb{R}^{m_1} \times \mathbb{R}^{m_1 \times m_0} \times \mathbb{R}^{m_0} \to \mathbb{R},$$

given by

$$G(B, b, \gamma, \beta, A, x_\alpha) = \mathbb{L}_y \circ g^{[2]} \circ \Phi^{[2]}(B, b, g^{[1]} \circ BN_\alpha^{\gamma,\beta}(\Phi^{[1]}(A, x))).$$

We compute (and since $\alpha \in \{1, ..., N\}$ is fixed, we ignore implied summation for the moment)

- $$\begin{aligned} d_B G_B(V) &= d_B(\mathbb{L}_y \circ g^{[2]} \circ \Phi^{[2]})_B(V) \\ &= \left.\frac{d}{dt}\right|_{t=0} \mathbb{L}_y \circ g^{[2]}((B + tV){a^{[1]}}_\alpha + b) \\ &= ({\delta^{[2]}}_\alpha{}^T)_\rho \left.\frac{d}{dt}\right|_{t=0} \left[(B^\rho_\lambda + tV^\rho_\lambda){a^{[1]}}^\lambda_\alpha + b^\rho)\right] \\ &= ({\delta^{[2]}}_\alpha{}^T)_\rho V^\rho_\lambda {a^{[1]}}^\lambda_\alpha \\ &= ({a^{[1]}}_\alpha {\delta^{[2]}}_\alpha{}^T)^\lambda_\rho V^\rho_\lambda, \end{aligned}$$

  and hence

  $$d_B G_B = {a^{[1]}}_\alpha {\delta^{[2]}}_\alpha{}^T, \qquad \frac{\partial G}{\partial B} = {\delta^{[2]}}_\alpha {a^{[1]}}_\alpha{}^T.$$

- $$\begin{aligned} d_b G_b(v) &= d_B(\mathbb{L}_y \circ g^{[2]} \circ \Phi^{[2]})_b(v) \\ &= ({\delta^{[2]}}_\alpha{}^T)_\rho \left.\frac{d}{dt}\right|_{t=0} \left[B^\rho_\lambda {a^{[1]}}^\lambda_\alpha + (b^\rho + tv^\rho)\right] \\ &= {\delta^{[2]}}_\alpha{}^T v \end{aligned}$$

  yielding

  $$d_b G_b = {\delta^{[2]}}_\alpha{}^T, \qquad \frac{\partial G}{\partial b} = {\delta^{[2]}}_\alpha.$$

- $$\begin{aligned} d_\gamma G_\gamma(\xi) &= d_\gamma(\mathbb{L}_y \circ g^{[2]} \circ \Phi^{[2]}(B, b, g^{[1]} \circ BN^{\beta, {z^{[1]}}_\alpha}_\alpha))_\gamma(\xi) \\ &= ({\delta^{[2]}}_\alpha{}^T) \cdot B \cdot dg^{[1]}_{{\tilde{z}^{[1]}}_\alpha}(\hat{z}_\alpha \odot \xi) \\ &= ({\delta^{[2]}}_\alpha{}^T) \cdot B \cdot dg^{[1]}_{{\tilde{z}^{[1]}}_\alpha} \mathrm{diag}(\hat{z}^{[1]}_\alpha)\xi \\ &= {\delta^{[1]}}_\alpha{}^T \mathrm{diag}({\hat{z}^{[1]}}_\alpha)\xi, \end{aligned}$$

  and so

  $$d_\gamma G_\gamma = {\delta^{[1]}}_\alpha{}^T \mathrm{diag}({\hat{z}^{[1]}}_\alpha), \qquad \frac{\partial G}{\partial \gamma} = \mathrm{diag}({\hat{z}^{[1]}}_\alpha){\delta^{[1]}}_\alpha.$$

- $$\begin{aligned} d_\beta G_\beta(\eta) &= d_\beta(\mathbb{L}_y \circ g^{[2]} \circ \Phi^{[2]}(B, b, g^{[1]} \circ BN_\alpha^{\gamma, {z^{[1]}}_\alpha}))_\beta(\eta) \\ &= {\delta^{[1]}}_\alpha{}^T \eta, \end{aligned}$$

  thus
  $$d_\beta G_\beta = {\delta^{[1]}}_\alpha{}^T, \qquad \frac{\partial G}{\partial \beta} = {\delta^{[1]}}_\alpha.$$

- $$\begin{aligned} d_A G_A(V) &= {\delta^{[1]}}_\alpha{}^T \cdot d_\alpha(BN_\alpha^{\gamma,\beta})_{{z^{[1]}}_\alpha} d\Phi_A^{[1]}(V) \\ &= {\delta^{[1]}}_\alpha{}^T \mathrm{diag}(\gamma) d_\alpha(\mathcal{N}_\alpha)_{{z^{[1]}}_\alpha} V x_\alpha, \end{aligned}$$

  and hence
  $$\begin{aligned} d_A G_A &= x_\alpha {\delta^{[1]}}_\alpha{}^T \mathrm{diag}(\gamma) d_\alpha(\mathcal{N}_\alpha)_{{z^{[1]}}_\alpha}, \\ \frac{\partial G}{\partial A} &= \mathrm{diag}(\gamma) d_\alpha(\mathcal{N}_\alpha)_{{z^{[1]}}_\alpha} {\delta^{[1]}}_\alpha {x_\alpha}^T. \end{aligned}$$

Finally, since

$$\mathbb{J}(W^{[2]}, b^{[2]}, \gamma, \beta, W^{[1]}) = \frac{1}{N} \sum_{\alpha=1}^{N} G(W^{[2]}, b^{[2]}, \gamma, \beta, W^{[1]}, x_\alpha),$$

we've described our desired gradients after summation.

## 9.2 Inferencing

We note that in our computation for forward propagation, that our normalization transforms change with out batches. This leads to ambiguity when predicting a label for a new example. One fix would be to average our means and variances over our batches. That is, suppose during our iteration process, we have training-batches of the form $\{\mathbb{X}^k : 1 \leq k \leq K\}$, where each $\mathbb{X}^k$ has cardinality $|\mathbb{X}^k| = n$. Then for each hidden-layer $\ell \in \{1, ..., L-1\}$, we obtain the means

$${\mu^{[\ell]}}_k = \frac{1}{n} \sum_{x \in \mathbb{X}^k} z^{[\ell]},$$

and the variances

$${\sigma^{2[\ell]}}_k = \frac{1}{n} \sum_{x \in \mathbb{X}^k} (z^{[\ell]} - {\mu^{[\ell]}}_k)^2.$$

That is, for each hidden-layer $\ell$, we have the collection

$$\{\mu^{[\ell]}{}_k : 1 \leq k \leq K\}$$

from which we average again to obtain

$$\mu^{[\ell]} := \frac{1}{K} \sum_{k=1}^{K} \mu^{[\ell]}{}_k,$$

and the collection

$$\{\sigma^{2[\ell]}{}_k : 1 \leq k \leq K\},$$

from which we use the unbiased estimate

$$\sigma^{2[\ell]} := \frac{n}{n-1} \frac{1}{K} \sum_{k=1}^{K} \sigma^{2[\ell]}{}_k.$$

These quantities are what we use when computing the batch-normalization transforms of the hidden units for new examples.

## 9.3 Algorithm Outline

Suppose we have a training set $\mathbb{X}$ with which we wish to train a binary classification via an $L$-layer neural network. Let $N = |\mathbb{X}|$ and let $n = 2^p$ be the batch size with $K = \lceil \frac{N}{n} \rceil$ batches per epoch. Then our algorithm would be as follows:

1. Set hyper-parameters. Initialize parameters.

2. For $0 \leq i \leq$ `num_iters`:

   a. Generate batches $\{\mathbb{X}^k : 1 \leq k \leq K\}$.

   b. For $1 \leq k \leq K$:

      i. Perform forward propagation on $\mathbb{X}^k$:

      - $$z^{[1]} = W^{[1]}x$$

      - For $\ell \in \{1, ..., L-1\}$:

        - $$z^{[\ell]} = W^{[\ell]}a^{[\ell-1]}$$

        - $$\mu^{[\ell]}{}_k = \frac{1}{n}\sum_{x\in\mathbb{X}^k} z^{[\ell]}$$
        - $$\sigma^{2[\ell]}{}_k = \frac{1}{n}\sum_{x\in\mathbb{X}^k} (z^{[\ell]} - \mu^{[\ell]}{}_k)^2$$
        - $$\hat{z}^{[\ell]} = (\sigma^{2[\ell]}{}_k + \epsilon)^{-\frac{1}{2}} \odot (z^{[\ell]} - \mu^{[\ell]}{}_k)$$
        - $$\tilde{z}^{[\ell]} = \gamma^{[\ell]} \odot \hat{z}^{[\ell]} + \beta^{[\ell]}$$
        - $$a^{[\ell]} = g^{[\ell]}(\tilde{z}^{[\ell]})$$
    - $$z^{[L]} = W^{[L]}a^{[L-1]} + b$$
    - $$a^{[L]} = g^{[L]}(z^{[L]})$$

   ii. Compute cost $\mathbb{J}$ on $\mathbb{X}^k$.

   iii. Apply backwards propagation on $\mathbb{X}^k$ to obtain
   $$\frac{\partial \mathbb{J}}{\partial W^{[\ell]}}, \quad \frac{\partial \mathbb{J}}{\partial b}, \quad \frac{\partial \mathbb{J}}{\partial \gamma^{[\ell]}}, \quad \frac{\partial \mathbb{J}}{\partial \beta^{[\ell]}}.$$

   iv. Update parameters.

3. Compute
$$\mu^{[\ell]} = \mathbb{E}[\mu^{[\ell]}{}_k],$$
$$\sigma^{2[\ell]} = \frac{n}{n-1}\mathbb{E}[\sigma^{2[\ell]}{}_k]$$

4. Return
$$W^{[\ell]}, \quad b, \quad \gamma^{[\ell]}, \quad \beta^{[\ell]}, \quad \mu^{[\ell]}, \quad \sigma^{2[\ell]}.$$

## 9.4 Better Backpropagation

We consider a neural network utilizing batch normalization of the form

$$\begin{tikzcd}
& \mathbb{R}^{n^{[1]}\times n^{[0]}} \arrow[d] & & \mathbb{R}^{n^{[1]}} \arrow[d] & \mathbb{R}^{n^{[1]}} \arrow[d] & & \\
\mathbb{R}^{n^{[0]}\times N} \arrow[r] & \boxed{\phi^{[1]}} \arrow[r] & \boxed{N^{[1]}} \arrow[r] & \boxed{\Gamma^{[1]}} \arrow[r] & \boxed{\psi^{[1]}} \arrow[r] & \boxed{G^{[1]}} \arrow[r] & \cdots \\
& \cdots \arrow[r] & \boxed{\phi^{[2]}} \arrow[r] & \boxed{\psi^{[2]}} \arrow[r] & \boxed{G^{[2]}} \arrow[r] & \boxed{\mathbb{L}} \arrow[r] & \mathbb{R} \\
& & \mathbb{R}^{n^{[2]}\times n^{[1]}} \arrow[u] & \mathbb{R}^{n^{[2]}} \arrow[u] & & \mathbb{R}^{n^{[2]}\times N} \arrow[u] &
\end{tikzcd}$$

where we have the functions

1. $$\mathbb{L} : \mathbb{R}^{n^{[2]}\times N} \times \mathbb{R}^{n^{[2]}\times N} \to \mathbb{R}$$

   is the given loss function. If we're working with a binary classification problem, then we have that

   $$\begin{aligned}\mathbb{L}(y, \hat{y}) &= -\frac{1}{N}\sum_{j=1}^{n}\{y_j \log \hat{y}_j + (1-y_j)\log(1-\hat{y}_j)\} \\ &= -\frac{1}{n}\left[\langle y, \log y\rangle_{\mathbb{R}^N} + \langle 1-y, \log(1-\hat{y})\rangle_{\mathbb{R}^N}\right].\end{aligned}$$

2. $$G^{[\ell]} : \mathbb{R}^{n^{[\ell]}\times N} \to \mathbb{R}^{n^{[\ell]}\times N}$$

   is the broadcasting of the activation unit $g^{[\ell]} : \mathbb{R} \to \mathbb{R}$.

3. $$\phi^{[\ell]} : \mathbb{R}^{n^{[\ell]}\times n^{[\ell-1]}} \times \mathbb{R}^{n^{[\ell-1]}\times N} \to \mathbb{R}^{n^{[\ell]}\times N}$$

   is given by

   $$\phi^{[\ell]}(W, x) = Wx.$$

4. $$\psi^{[\ell]} : \mathbb{R}^{n^{[\ell]}} \times \mathbb{R}^{n^{[\ell]}\times N} \to \mathbb{R}^{n^{[\ell]}\times N}$$

   is given by

   $$\psi(b, x) = x + b\vec{1}^T,$$

   where

   $$\vec{1}^T = \begin{bmatrix}1 & 1 & \cdots & 1\end{bmatrix} \in \mathbb{R}^{n^{[\ell]}}.$$

5. $$N^{[1]} : \mathbb{R}^{n^{[1]}\times N} \to \mathbb{R}^{n^{[1]}\times N}$$
   is the normalization operator given by
   $$N^{[1]} : x^i_j \mapsto \frac{x^i_j - \mathbb{E}[x^i]}{\sqrt{\mathbb{V}[x^i] + \epsilon}},$$
   where $\mathbb{E}$ is the expectation operator, i.e.,
   $$\mathbb{E}[x^i] = \frac{1}{N}\sum_{j=1}^{N} x^i_j,$$
   and $\mathbb{V}$ is the variance operator, i.e.,
   $$\mathbb{V}[x^i] = \mathbb{E}[(x^i - \mathbb{E}[x^i])^2].$$

6. $$\Gamma^{[\ell]} : \mathbb{R}^{n^{[\ell]}} \times \mathbb{R}^{n^{[\ell]}\times N} \to \mathbb{R}^{n^{[\ell]}\times N}$$
   is given by
   $$\Gamma(\gamma, x) = \gamma\vec{1}^T \odot x,$$
   where
   $$\vec{1}^T = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} \in \mathbb{R}^{n^{[\ell]}}.$$

We now consider back-propagating through the network via reverse differentiations as in the following diagram:

$$\begin{tikzcd}
 & \mathbb{R}^{n^{[1]}\times n^{[0]}} & & \mathbb{R}^{n^{[1]}} & \mathbb{R}^{n^{[1]}} & & \\
\mathbb{R}^{n^{[0]}\times N} & \boxed{\phi^{[1]}} \arrow[l, "r"'] \arrow[u, "r_1"] & \boxed{N^{[1]}} \arrow[l, "r"'] & \boxed{\Gamma^{[1]}} \arrow[l, "r"'] \arrow[u, "\hat{r}_1"] & \boxed{\psi^{[1]}} \arrow[l, "r"'] \arrow[u, "\tilde{r}_1"] & \boxed{G^{[1]}} \arrow[l, "r"'] & \cdots \arrow[l, "r"'] \\
 & \cdots & \boxed{\phi^{[2]}} \arrow[l, "r"'] \arrow[d, "r_2"] & \boxed{\psi^{[2]}} \arrow[l, "r"'] \arrow[d, "\overline{r}_2"] & \boxed{G^{[2]}} \arrow[l, "r"'] & \boxed{\mathbb{L}} \arrow[l, "r"'] & \mathbb{R} \arrow[l] \\
 & & \mathbb{R}^{n^{[2]}\times n^{[1]}} & \mathbb{R}^{n^{[2]}} & & \mathbb{R}^{n^{[2]}\times N} \arrow[u] &
\end{tikzcd}$$

We consider our individual derivatives:

1. Suppose $G : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$ is the broadcasting of $g : \mathbb{R} \to \mathbb{R}$. Then for any $(x, \xi) \in T\mathbb{R}^{m\times n}$ we have that
   $$dG_x(\xi) = G'(x) \odot \xi.$$
   Then for any $\zeta \in T_{G(x)}\mathbb{R}^{m\times n}$, we have the reverse derivative is given by
   $$rG_x(\zeta) = G'(x) \odot \zeta.$$

2. Suppose $\phi : \mathbb{R}^{m\times n} \times \mathbb{R}^{n\times N} \to \mathbb{R}^{m\times N}$ is given by

   $$\phi(W, x) = Wx.$$

   Then we have two differential paths to consider:

   (a) For any $(W, x) \in \mathbb{R}^{m\times n} \times \mathbb{R}^{n\times N}$ and any $\xi \in T_x\mathbb{R}^{n\times N}$, we have that

   $$\begin{aligned} d\phi_{(W,x)}(\xi) &= W \cdot \xi \\ &= L_W(\xi), \end{aligned}$$

   and for any $\zeta \in T_{\phi(W,x)}\mathbb{R}^{m\times N}$, we have the reverse differential

   $$\begin{aligned} r\phi_{(W,x)}(\zeta) &= W^T \cdot \zeta \\ &= L_{W^T}(\zeta). \end{aligned}$$

   (b) For any $(W, x) \in \mathbb{R}^{m\times n} \times \mathbb{R}^{n\times N}$ and any $Z \in T_W\mathbb{R}^{m\times n}$, we have that

   $$\begin{aligned} d_1\phi_{(W,x)}(Z) &= Z \cdot x \\ R_x(Z)&, \end{aligned}$$

   and for any $\zeta \in T_{\phi(W,x)}\mathbb{R}^{m\times N}$, we have the reverse differential

   $$\begin{aligned} r_1\phi_{(W,x)}(\zeta) &= \zeta \cdot x^T \\ &= R_{x^T}(\zeta). \end{aligned}$$

3. Suppose $\psi : \mathbb{R}^n \times \mathbb{R}^{n\times N} \to \mathbb{R}^{n\times N}$ is given by

   $$\psi(b, x) = x + b\vec{1}^T,$$

   where

   $$\vec{1}^T = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} \in \mathbb{R}^N.$$

   Then we look at the two differential paths and for any $(b, x) \in \mathbb{R}^n \times \mathbb{R}^{n\times N}$ any any $\xi \in T_x\mathbb{R}^{n\times N}$, $\eta \in T_b\mathbb{R}^n$ and $\zeta \in T_{\psi(b,x)}\mathbb{R}^{n\times N}$:

   (a) In the network direction, we have that

   $$d\psi_{(b,x)}(\xi) = \xi,$$

   with reverse differential

   $$r\psi_{(b,x)}(\zeta) = \zeta.$$

(b) In the parameter-space direction, we have that

$$\begin{aligned}\overline{d}\psi_{(b,x)}(\eta) &= \eta \cdot \vec{1}^T \\ &= R_{\vec{1}^T}(\eta),\end{aligned}$$

with reverse differential

$$\begin{aligned}\overline{r}\psi_{(b,x)}(\zeta) &= \zeta \cdot \vec{1} \\ &= R_{\vec{1}}(\zeta).\end{aligned}$$

4. Suppose $\Gamma : \mathbb{R}^n \times \mathbb{R}^{n\times N} \to \mathbb{R}^{n\times N}$ is given by

$$\Gamma(\gamma, x) = \gamma\vec{1}^T \odot x.$$

The considering the two paths of differentiation, we have that for any $((\gamma, x), (\eta, \xi)) \in T(\mathbb{R}^n \times \mathbb{R}^{n\times N})$ and $\zeta \in T_{\Gamma(\gamma,x)}\mathbb{R}^{n\times N}$ that:

(a) In the network direction, we have that

$$d\Gamma_{(\gamma,x)}(\xi) = \gamma\vec{1}^T \odot \xi,$$

with reverse differential

$$r\Gamma_{(\gamma,x)}(\zeta) = \gamma\vec{1}^T \odot \zeta.$$

(b) In the parameter-space direction, we have that

$$\begin{aligned}\hat{d}\Gamma_{(\gamma,x)}(\eta) &= \eta\vec{1}^T \odot x \\ &= \odot_x \circ R_{\vec{1}^T}(\eta),\end{aligned}$$

with reverse differential

$$\begin{aligned}\hat{r}\Gamma_{(\gamma,x)}(\zeta) &= (x \odot \zeta) \cdot \vec{1} \\ &= R_{\vec{1}} \circ \odot_x(\zeta).\end{aligned}$$

5. As the normalization operator is quite involved, we move its computation to the appendix, Section C.

6. For the loss function $\mathbb{L} : \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}$ given by

$$L(y, \hat{y}) = -\frac{1}{N}[\langle y, \log \hat{y}\rangle + \langle 1 - y, \log(1 - \hat{y})\rangle],$$

we fix $y, \hat{y} \in \mathbb{R}^N$ and for $\xi \in T_{\hat{y}}\mathbb{R}^N$, we see that

$$\begin{aligned} d\mathbb{L}_{(y,\hat{y})}(\xi) &= -\frac{1}{N}\sum_{j=1}^{N}\left[\frac{y_j}{\hat{y}_j} - \frac{1-y_j}{1-\hat{y}_j}\right]\xi_j \\ &= -\frac{1}{N}\left\langle \frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}}, \xi \right\rangle, \end{aligned}$$

and hence for $\zeta \in T_{L(y,\hat{y})}\mathbb{R}$, it follows that

$$r\mathbb{L}_{(y,\hat{y})}(\zeta) = -\frac{1}{N}\left[\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}}\right]\zeta.$$

We're now ready to compute our various gradients of our cost function. That is, if we let

$$\mathbb{J} : \mathbb{R}^{n^{[2]}} \times \mathbb{R}^{n^{[2]} \times n^{[1]}} \times \mathbb{R}^{n^{[1]}} \times \mathbb{R}^{n^{[1]}} \times \mathbb{R}^{n^{[1]} \times n^{[0]}} \to \mathbb{R}$$

is given by

$$\mathbb{J}(W^{[2]}, \gamma^{[1]}, \beta^{[1]}, W^{[2]}, b^{[2]}) = \mathbb{L}(y, G^{[2]} \circ \psi^{[2]}(b^{[2]}, \phi^{[2]}(W^{[2]}, G^{[2]} \circ \psi^{[2]}(\beta^{[1]}, \Gamma^{[1]}(\gamma^{[1]}, N^{[1]} \circ \phi^{[1]}(W^{[1]}, x))))))$$

and we compute the reverse differentials for a learning rate $\alpha \in T_{\mathbb{J}}\mathbb{R}$ with the assumption that our second activator function is the sigmoid function. Indeed,

$$\begin{aligned} r(\mathbb{L} \circ G^{[2]})_v(\alpha) &= rG^{[2]}_v \circ r\mathbb{L}_a(\alpha) \\ &= -\frac{\alpha}{N}G^{[2]\prime}(v) \odot \left[\frac{y}{a} - \frac{1-y}{1-a}\right] \\ &= -\frac{\alpha}{N}a(1-a)\left[\frac{y}{a} - \frac{1-y}{1-a}\right] \\ &= -\frac{\alpha}{N}[y(1-a) - a(1-y)] \\ &= -\frac{\alpha}{N}[y-a] \\ &= \frac{a-y}{N}\alpha. \end{aligned}$$

This leads us to

$$\begin{aligned}
\overline{r}_2\mathbb{J}_{b^{[2]}}(\alpha) &= \overline{r}_2(\psi^{[2]})_{(b^{[2]},u^{[2]})} \circ rG^{[2]}_{v^{[2]}} \circ r\mathbb{L}_{(y,a^{[2]})} \\
&= \frac{\alpha}{N} R_{\vec{1}}(a^{[2]} - y) \\
&= \frac{\alpha}{N} \sum_{j=1}^{N} ({a^{[2]}}_j - y_j);
\end{aligned}$$

$$\begin{aligned}
r_2\mathbb{J}_{W^{[2]}}(\alpha) &= r_2\phi^{[2]}_{(W^{[2]},a^{[1]})} \circ r\psi^{[2]}_{(b^{[2]},u^{[2]})} \left(\frac{\alpha}{N}(a^{[2]} - y)\right) \\
&= r_2\phi^{[2]}_{(W^{[2]},a^{[1]})} \left(\frac{\alpha}{N}(a^{[2]} - y)\right) \\
&= \frac{\alpha}{N}(a^{[2]} - y)a^{[1]T};
\end{aligned}$$

$$\begin{aligned}
\overline{r}_1\mathbb{J}_{\beta^{[1]}}(\alpha) &= \overline{r}_1\psi^{[1]}_{(\beta^{[1]},\hat{z}^{[1]})} \circ rG^{[1]}_{\tilde{z}^{[1]}} \circ r\phi^{[2]}_{(W^{[2]},a^{[2]})} \circ r\psi^{[2]}_{(b^{[2]},u^{[2]}} \circ r(\mathbb{L} \circ G^{[2]})_{v^{[2]}}(\alpha) \\
&= \frac{\alpha}{N}\overline{r}_1\psi^{[1]}_{(\beta^{[1]},\hat{z}^{[1]})} \circ rG^{[1]}_{\tilde{z}^{[1]}} \circ r\phi^{[2]}_{(W^{[2]},a^{[2]})}(a^{[2]} - y) \\
&= \frac{\alpha}{N}\overline{r}_1\psi^{[1]}_{(\beta^{[1]},\hat{z}^{[1]})} \circ rG^{[1]}_{\tilde{z}^{[1]}} \left(W^{[2]T}(a^{[2]} - y)\right) \\
&= \frac{\alpha}{N}\overline{r}_1\psi^{[1]}_{(\beta^{[1]},\hat{z}^{[1]})} \left(G^{[1]\prime}(\tilde{z}^{[1]}) \odot W^{[2]T}(a^{[2]} - y)\right) \\
&= \frac{\alpha}{N} \sum_{j=1}^{N} g^{[1]\prime}({\tilde{z}^{[1]}}_j) \odot W^{[2]T}({a^{[2]}}_j - y_j);
\end{aligned}$$

$$\begin{aligned}
\hat{r}_1\mathbb{J}_{\gamma^{[1]}}(\alpha) &= \frac{\alpha}{N}\hat{r}_1\Gamma^{[1]}_{(\gamma^{[1]},z^{[1]})} \left(G^{[1]\prime}(\tilde{z}^{[1]}) \odot W^{[2]T}(a^{[2]} - y)\right) \\
&= \frac{\alpha}{N} R_{\vec{1}} \left(z^{[1]} \odot \left(G^{[1]\prime}(\tilde{z}^{[1]}) \odot W^{[2]T}(a^{[2]} - y)\right)\right) \\
&= \frac{\alpha}{N} \sum_{j=1}^{n} {z^{[1]}}_j \odot g^{[1]\prime}({\tilde{z}^{[1]}}_j) \odot W^{[2]T}({a^{[2]}}_j - y_j);
\end{aligned}$$

and finally,

$$\begin{aligned} r_1\mathbb{J}_{W^{[1]}}(\alpha) &= \frac{\alpha}{N} r_1\phi^{[1]}_{(W^{[1]},x)} \circ rN^{[1]}_{u^{[1]}} \circ r\Gamma^{[1]}_{(\gamma^{[1]},z^{[1]})} \left(G^{[1]\prime}(\tilde{z}^{[1]}) \odot W^{[2]T}(a^{[2]} - y)\right) \\ &= \frac{\alpha}{N} r_1\phi^{[1]}_{(W^{[1]},x)} \circ rN^{[1]}_{u^{[1]}} \left(\gamma\vec{1}^T \odot G^{[1]\prime}(\tilde{z}^{[1]}) \odot W^{[2]T}(a^{[2]} - y)\right) \\ &= \frac{\alpha}{N} R_{x^T} \circ rN^{[1]}_{u^{[1]}} \left(\gamma\vec{1}^T \odot G^{[1]\prime}(\tilde{z}^{[1]}) \odot W^{[2]T}(a^{[2]} - y)\right) \\ &= \frac{\alpha}{N} \sum_{j,l=1}^{N} \sum_{i=1}^{n^{[1]}} T_i{}^{jk}{}_l \gamma^i g^{[1]\prime}(\tilde{z}^{[1]i}{}_j) W^{[2]}{}_i (a^{[2]}{}_j - y_j) x_l^m \end{aligned}$$

## 9.5 Python Implementation

Work in Progress

# 10 Multi-Class Softmax Regression

Thus far, we've mostly been dealing with binary classification problems, that is, our true label $y$ takes values in $\{0,1\}$, where $y=1$ represents when the object in question represents our desired classification, and $y=0$ when it does not. However, in many examples we wish to expand upon this, for example, instead of knowing whenever an image contains a cat ($y=1$) or it doesn't contain a cat ($y=0$), maybe we would like to have a table of the following

Table 1: Classification

| $y$ | Label |
|---|---|
| $y=0$ | None of the following |
| $y=1$ | Cat |
| $y=2$ | Dog |
| $y=3$ | Bird |
| $y=4$ | Elephant |
| $y=5$ | Bear |

That is, we have a total of 6 classes we wish to distinguish. If we were to train a neural network for this classification problem, the only time this needs to be considered is on the output layer. With this in mind, we shall only consider the simple regression problem

$$\begin{bmatrix} x^1 \\ \vdots \\ x^m \end{bmatrix} \overbrace{\longrightarrow}^{Wx+b} \begin{bmatrix} z^1 \\ \vdots \\ z^C \end{bmatrix} \overbrace{\longrightarrow}^{g(z)} \begin{bmatrix} a^1 \\ \vdots \\ a^C \end{bmatrix} \longrightarrow \hat{y},$$

where $C$ is the number of labels in our classification.

First, we need to *one-hot encode* our labels. That is, if our labels are given by

$$\{0,1,...,C-1\},$$

then we consider the basis vectors in $\mathbb{R}^C$

$$\{e_1,...,e_C\},$$

which clearly admits a bijection

$$\{0,1,...,C-1\} \xrightarrow{\cong} \{e_1,...,e_C\}, \qquad i \mapsto e_{i+1}.$$

Thus, we've effectively mapped our true labels

$$y \in \{0, 1, ..., C-1\}^N \mapsto y \in \mathbb{R}^{C \times N},$$

where

$$(y = i) \mapsto (y = e_{i+1}).$$

Next, we need to decide which type of nonlinearity $g : \mathbb{R}^C \to \mathbb{R}^C$ to impose. To this end, we would like $a^i$ to satisfy

$$a^i = \mathbb{P}(y = i - 1),$$

then we can declare a prediction via

$$i_0 = \arg\max_i a^i, \qquad \hat{y} = e_{i_0} \leftrightarrow \hat{y} = i_0 - 1.$$

That is, we would like our target output vector $a \in \mathbb{R}^C$ to be a probability distribution, i.e.,

$$0 \leq a^i \leq 1, i \in \{1, ..., C\},$$

and

$$\sum_{i=1}^{C} a^i = 1.$$

This leads us to letting $g$ be the softmax function, i.e.,

$$g(z^1, ..., z^C) = \frac{1}{\sum_{i=1}^{C} e^{z^i}} \begin{bmatrix} e^{z^1} \\ \vdots \\ e^{z^C} \end{bmatrix}.$$

Finally, we need to define a cost function $\mathbb{L} : \mathbb{R}^C \times \mathbb{R}^C \to \mathbb{R}$ with which we can compare our true value to our predicted value. To this end, we consider the cross-entropy function $\mathbb{L}$ defined by

$$\mathbb{L}(a_j, y_j) = -\sum_{i=1}^{C} y_j^i \log a_j^i.$$

We note that since $y_j = e_k$ for some $k \in \{1, ..., C\}$, that this sum is actually a single element. Moreover, when $C = 2$, we recover our log-loss function for the sigmoid activation. This finally yields a cost function

$$\begin{aligned} \mathbb{J}(W, b) &= -\frac{1}{N} \sum_{j=1}^{N} \sum_{i=1}^{C} y_j^i \log a_j^i \\ &= -\frac{1}{N} (y : \log a), \end{aligned}$$

where

$$A : B = \langle A, B \rangle_F = \operatorname{tr}(A^T B),$$

is the Frobenius norm on $\mathbb{R}^{C \times N}$.

To minimize our cost, we first note

$$\begin{aligned}
\frac{\partial \mathbb{L}_y \circ g}{\partial z^\mu} &= \sum_{i=1}^{C} \frac{\partial \mathbb{L}_y}{\partial a^i} \frac{\partial S^i}{\partial z^\mu} \\
&= -\sum_{i=1}^{C} \frac{y^i}{a^i} a^i (\delta^i_\mu - a^\mu) \\
&= -\sum_{i=1}^{C} y^i (\delta^i_\mu - a^\mu) \\
&= -y^\mu + a^\mu \underbrace{\sum_{i=1}^{C} y^i}_{=1} \\
&= a^\mu - y^\mu,
\end{aligned}$$

then we see that

$$\begin{aligned}
\frac{\partial z^\mu}{\partial W^\alpha_\beta} &= \frac{\partial}{\partial W^\alpha_\beta} (W^\mu_k x^k + b^\mu) \\
&= \sum_{k=1}^{m} \delta^\mu_\alpha \delta^\beta_k x^k \\
&= \delta^\mu_\alpha x^\beta,
\end{aligned}$$

and

$$\frac{\partial z^\mu}{\partial b^\alpha} = \delta^\mu_\alpha.$$

Hence,

$$\begin{aligned}
\frac{\partial \mathbb{L}_y}{\partial W^\alpha_\beta} &= \sum_{\mu=1}^{C} (a^\mu - y^\mu) \delta^\mu_\alpha x^\beta \\
&= x(a-y)^T,
\end{aligned}$$

yielding a gradient of

$$\frac{\partial \mathbb{L}_y}{\partial W} = (a-y)x^T,$$

and similarly

$$\begin{aligned}\frac{\partial \mathbb{L}_y}{\partial b^\alpha} &= \sum_{\mu=1}^{C}(a^\mu - y^\mu)\delta^\mu_\alpha \\ &= a^\alpha - y^\alpha,\end{aligned}$$

and so

$$\frac{\partial \mathbb{L}_y}{\partial b} = a - y.$$

Finally, we conclude that

$$\frac{\partial \mathbb{J}}{\partial W} = \frac{1}{N}\sum_{j=1}^{N}(a_j - y_j)(x_j)^T = \frac{1}{N}(a-y)x^T,$$

and

$$\frac{\partial \mathbb{J}}{\partial b} = \frac{1}{N}\sum_{j=1}^{N}(a_j - y_j).$$

We remark that for a deep neural network, the backwards propagation follows a similar path backwards through the network since we have the aforementioned differentials.

# Part III
# Convolutional Neural Networks

# 11 An Introduction to Convolutions

One common application of neural networks is that of image detection/-classification. Recall that an image in grayscale can be seen as a matrix $x \in \mathbb{R}^{m\times n}$, where

$$x^i_j \in \{0, 1, ..., 9, 10\},$$

and 10 represents "white" and 0 represents "black".

Instead of flattening the pixels into a vector $\vec{x} \in \mathbb{R}^{nm}$ and feeding the input into a deep network, we observe that several simple detections may be imposed on the image first while it's in matrix form. That is, suppose we wish to detect vertical or horizontal edges in the image first. As there are typically several of such edges in an image, and these edges are the "atomic" pieces of full images, this initial detection would be of great benefit.

To this end, we wish to impose an operation which finds where a pixel $x^i_j$ changes dramatically when moving to a neighboring pixel. One way to find these changes is with convolutions, or cross-correlations.

## 11.1 Cross-Correlation

We first recall that given two function $f, g : \mathbb{Z} \to \mathbb{R}$, the (discrete) cross-correlation $f * g$ is defined by

$$f * g(n) = \sum_{j=-\infty}^{\infty} f(j)g(j+n).$$

We note that cross-correlation is not commutative, however, we see that

$$\begin{aligned} g * f(-n) &= \sum_{j=-\infty}^{\infty} g(j)f(j-n) \qquad i = j - n \\ &= \sum_{i=-\infty}^{\infty} f(i)g(i+n) \\ &= f * g(n). \end{aligned}$$

We may similarly define for $f, g : \mathbb{Z}^2 \to \mathbb{R}$,

$$f * g(k, l) = \sum_{(i,j)\in\mathbb{Z}^2} f(i, j)g(i + k, j + l).$$

Whenever $f$ or $g$ has finite support, say in $[-M, M]$, the above sum reduces to

$$f * g(n) = \sum_{j=-M}^{M} f(j)g(j+n).$$

Suppose $x \in \mathbb{R}^{{n_h}^{[0]} \times {n_w}^{[0]}}$ and let $F \in \mathbb{R}^{f^{[1]} \times f^{[1]}}$ with $f^{[1]} \leq \min\{{n_h}^{[0]}, {n_w}^{[0]}\}$. Define

$${n_\alpha}^{[1]} = {n_\alpha}^{[0]} - f^{[1]} + 1, \qquad \alpha = h, w,$$

and we obtain the matrix $F * x \in \mathbb{R}^{{n_h}^{[1]} \times {n_w}^{[1]}}$ given by

$$(F * x)^k_l = \sum_{i,j=1}^{f^{[1]}} F^i_j x^{i+k-1}_{j+l-1}.$$

Note that this is exactly the cross-correlation defined above, except with finite support and reindexed to start at 1.

In what follows, this cross-correlation operator will be called the *convolution* operator, and $F$ will be called the filter (or kernel).

**Example 11.1.** *Suppose*

$$x = \begin{bmatrix} 1 & 2 & 0 & 3 \\ 4 & 5 & 6 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix}$$

*and*

$$F = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

*Then* $f = 2$*,* ${n_h}^{[0]} = 3$*,* ${n_w}^{[0]} = 4$*, and so*

$${n_h}^{[1]} = 3 - 2 + 1 = 2,$$

$${n_w}^{[1]} = 4 - 2 + 1 = 3.$$

*We now compute* $(F * x) \in \mathbb{R}^{2\times 3}$

$$\begin{aligned}
(F * x)^1_1 &= 1 * 1 + 0 * 2 + 1 * 4 + 1 * 5 = 10 \\
(F * x)^1_2 &= 1 * 2 + 0 * 0 + 1 * 5 + 1 * 6 = 13 \\
(F * x)^1_3 &= 1 * 0 + 0 * 3 + 1 * 6 + 1 * 0 = 6 \\
(F * x)^2_1 &= 1 * 4 + 0 * 5 + 1 * 0 + 1 * 1 = 5 \\
(F * x)^2_2 &= 1 * 5 + 0 * 6 + 1 * 1 + 2 * 2 = 10 \\
(F * x)^2_3 &= 1 * 6 + 0 * 0 + 1 * 2 + 1 * 3 = 11,
\end{aligned}$$

*and hence*

$$F * x = \begin{bmatrix} 10 & 13 & 6 \\ 5 & 10 & 11 \end{bmatrix}.$$

**Example 11.2.** *Suppose*

$$x = \begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix},$$

*which can be seen as a grayscale image that's white on the left half of the image and black on the right half. Now define the filter*

$$F = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}.$$

*Then $F * x \in \mathbb{R}^{4\times 4}$ and is given by*

$$F * x = \begin{bmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{bmatrix},$$

*which looks like an image a "white" edge in the middle, telling us the original has an edge in the middle that goes from "bright" pixels to "dark" pixels.*

This idea of convolution seems to be able to detect our edges. However, we see that the pixels in the "interior" of the matrix affect the convolution much more the the pixels on the "boundary". This may not always matter, but when it does, we need a technique to allow for the boundary pixels to be more prominent. One such fix is to add some "padding" around the original image.

## 11.2 Convolution with Padding

Suppose $x \in \mathbb{R}^{m\times n}$ is matrix, and let $p \in \mathbb{Z}_{\geq 0}$, which we will call the *padding*. Define a new matrix $(x, p) \in \mathbb{R}^{(m+2p)\times(n+2p)}$ given by

$$(x,p)^k_l = \begin{cases} x^{k-p}_{l-p} & \text{if } p < k \leq m+p \text{ and } p < l \leq n+p, \\ 0 & \text{else.} \end{cases}$$

**Example 11.3.** *Suppose*

$$x = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}.$$

*Then* $(x, 0) = x$ *immediately,*

$$(x, 1) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

$$(x, 2) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 & 0 \\ 0 & 0 & 3 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} = ((x, 1), 1).$$

From the previous example, we see a recursive property with padding, i.e.,

$$\begin{aligned} (x, p) &= ((x, p-1), 1) \\ &= (((x, p-2), 1), 1) \\ &\vdots \\ &= \underbrace{((\cdots((}_{\text{p-times}} x, \underbrace{1), 1), \cdots 1), 1)}_{p-\text{times}} \end{aligned}$$

Suppose $x \in \mathbb{R}^{{n_h}^{[0]} \times {n_w}^{[0]}}$, let $F \in \mathbb{R}^{f^{[1]} \times f^{[1]}}$ be a filter, and let $p \in \mathbb{Z}_{\geq 0}$ be the padding. Then since $(x, p)$ is an $({n_h}^{[0]} + 2p) \times ({n_w}^{[0]} + 2p)$-matrix, we have that the convolution $F * (x, p)$ has a size given by

$${n_\alpha}^{[1]} = {n_\alpha}^{[0]} + 2p - f^{[1]} + 1, \qquad \alpha = h, w,$$

and we write

$$F *^p x = F * (x, p).$$

When $p = 0$, we say that $F *^p x$ is a *valid convolution*, and we'll typically drop the $p$-superscript. When $p = \frac{f^{[1]}-1}{2}$, we say that $F *^p x$ is a *same convolution*, since

$${n_\alpha}^{[1]} = {n_\alpha}^{[0]}, \qquad \alpha = h, w.$$

We remark here that in many application our desired filters have $f^{[1]}$ being odd (if it's not odd, then it cannot be a same convolution).

## 11.3 Strided Convolution

We note that in our definition of a convolution

$$(F * x)^k_l = \sum_{i,j=1}^{f^{[1]}} F^i_j x^{i+k-1}_{j+l-1},$$

that we're sliding our filter $F$ along $x$ with a *stride* of $s = 1$. This does not necessarily have to be the case. We modify our definition of convolution to allow for $s \in \mathbb{N}$ as follows:

Suppose $x \in \mathbb{R}^{{n_h}^{[0]} \times {n_w}^{[0]}}$, let $F \in \mathbb{R}^{f^{[1]} \times f^{[1]}}$ be a filter and let $s \in \mathbb{N}$ be the stride. Let

$$ {n_\alpha}^{[1]} = \lfloor \frac{{n_\alpha}^{[0]} - f^{[1]}}{s} + 1 \rfloor, \qquad \alpha = h, w,$$

and define $F *_s x \in \mathbb{R}^{{n_h}^{[1]} \times {n_w}^{[1]}}$ to be the matrix given by

$$(F *_s x)^k_l = \sum_{i,j=1}^{f^{[1]}} F^i_j x^{i+s(k-1)}_{j+s(l-1)}.$$

We note that the definition of a strided convolution is a direct generalization of our previous definition of convolution, namely with stride $s = 1$.

**Example 11.4.** *Suppose*

$$x = \begin{bmatrix} 1 & 0 & 2 & 0 \\ 3 & 0 & 4 & 0 \\ 0 & 5 & 0 & 6 \\ 7 & 0 & 8 & 0 \end{bmatrix},$$

$$F = \begin{bmatrix} 1 & 1 \\ 2 & 0 \end{bmatrix},$$

*and suppose we have a stride of* 2 *(any larger stride would result in a* $(1 \times 1)$*-matrix). Then we see that*

$$ {n_\alpha}^{[1]} = \lfloor \frac{4-2}{2} + 1 \rfloor = 2, \qquad \alpha = h, w,$$

*and hence that*

$$\begin{aligned}
(F *_2 x)^1_1 &= 1 * 1 + 1 * 0 + 2 * 3 + 0 * 0 = 7 \\
(F *_2 x)^1_2 &= 1 * 2 + 1 * 0 + 2 * 4 + 0 * 0 = 10 \\
(F *_2 x)^2_1 &= 1 * 0 + 1 * 5 + 2 * 7 + 0 * 0 = 19 \\
(F *_2 x)^2_2 &= 1 * 0 + 1 * 6 + 2 * 8 + 0 * 0 = 22,
\end{aligned}$$

*or rather*

$$F *_2 x = \begin{bmatrix} 7 & 10 \\ 19 & 22 \end{bmatrix}.$$

## 11.4 Strided Convolutions with Padding

Suppose $x \in \mathbb{R}^{{n_h}^{[0]} \times {n_w}^{[0]}}$, let $F \in \mathbb{R}^{f^{[1]} \times f^{[1]}}$ be a filter, let $s \in \mathbb{N}$ be the stride, and let $p \in \mathbb{Z}_{\geq 0}$ be the padding. We define

$$F *_s^p x := F *_s (x, p),$$

that is, we first pad $x$, then compute the strided convolution of the filter $F$ with $(x, p)$. From our previous work, we see that for $\alpha = h, w$, that

$$\begin{aligned} {n_\alpha}^{[1]} &= \left\lfloor \frac{{n'_\alpha}^{[0]} - f^{[1]}}{s} + 1 \right\rfloor, \qquad n' \sim (x, p) \\ &= \left\lfloor \frac{{n_\alpha}^{[0]} + 2p - f^{[1]}}{s} + 1 \right\rfloor. \end{aligned}$$

Moreover, to compute a closed form of the strided convolution with padding, we first define the set

$$\begin{aligned} \mathcal{I}^{[1]}{}^k_l &= \mathcal{I}({n_h}^{[0]}, {n_w}^{[0]}, p, s; k, l) \\ &:= \big\{(i, j) \in \mathbb{Z}^2 : p < i + s(k-1) - p \leq {n_h}^{[0]} + p\ ; \\ &\qquad\qquad p < j + s(l-1) - p \leq {n_w}^{[0]} + p\big\} \\ &= \big\{(i, j) \in \mathbb{Z}^2 : 2p - s(k-1) < i \leq 2p - s(k-1) + {n_h}^{[0]}\ ; \\ &\qquad\qquad 2p - s(l-1) < j \leq 2p - s(l-1) + {n_w}^{[0]}\big\} \end{aligned}$$

and now we immediately see by chasing the definitions that

$$\begin{aligned} (F *_s^p x)^k_l &= (F *_s (x, p))^k_l \\ &= \sum_{i,j=1}^{f^{[1]}} F^i_j (x, p)^{i+s(k-1)}_{j+s(l-1)} \\ &= \sum_{i,j=1}^{f^{[1]}} F^i_j x^{i+s(k-1)-p}_{j+s(l-1)-p} \chi_{\mathcal{I}^{[1]}{}^k_l}(i, j) \end{aligned}$$

**Example 11.5.** *Suppose*

$$x = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 3 & 0 \\ 4 & 0 & 5 \end{bmatrix},$$

*and we have a filter*

$$F = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

*We first compute $F *_2^1 x$: Since we we're using a padding of $p = 1$, we have that*

$$(x, 1) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 4 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

*Using a stride of $s = 2$, we see we have resultant dimensions of the form*

$$\begin{aligned} {n_\alpha}^{[1]} &= \lfloor \frac{3 + 2 * 1 - 2}{2} + 1 \rfloor \\ &= 2, \end{aligned}$$

*that is, $F *_2^1 x \in \mathbb{R}^{2\times 2}$. We now compute*

$$\begin{aligned} (F *_2^1 x)_1^1 &= 1 * 0 + 1 * 0 + 0 * 0 + 1 * 1 = 1 \\ (F *_2^1 x)_2^1 &= 1 * 0 + 1 * 0 + 0 * 0 + 1 * 2 = 2 \\ (F *_2^1 x)_1^2 &= 1 * 0 + 1 * 0 + 0 * 0 + 1 * 4 = 4 \\ (F *_2^1 x)_2^2 &= 1 * 0 + 1 * 0 + 0 * 5 + 1 * 0 = 0, \end{aligned}$$

*or rather*

$$F *_2^1 x = \begin{bmatrix} 1 & 2 \\ 4 & 0 \end{bmatrix}.$$

## 11.5 Convolutions Over Volumes

At the beginning of this section, we began by considering a grayscale image which we represented as a matrix $x \in \mathbb{R}^{n_h \times n_w}$. Suppose that instead of grayscale, we have an RGB image. Then for each fixed color component, we may represent the component as a matrix as before. However, since flattening a color image into a grayscale image would break our desired symmetries (e.g., for edges, etc), we would like a way to handle convolutions of an RGB image being represented as a rank-3 tensor $x \in \mathbb{R}^{n_h \times n_w \times n_c}$. This $n_c$ parameter represents the "depth" of the image, which we shall call the *channels*. That is, $x$ has a red, a green, and a blue channel. We wish to work with channels simultaneously to see simplifications in their relationships with each other. To this end, we introduce a notion of convolution over volumes, which instead of moving a $f^{[1]} \times f^{[1]}$-square across $x$, we move a $f^{[1]} \times f^{[1]} \times {n_c}^{[0]}$-prism across $x$ instead.

Suppose $x \in \mathbb{R}^{{n_h}^{[0]} \times {n_w}^{[0]} \times {n_c}^{[0]}}$, and suppose $F \in \mathbb{R}^{f^{[1]} \times f^{[1]} \times {n_c}^{[0]}}$ is a filter (noted the channel size of the input must match the channel size of the filter). Then as before we have that

$${n_\alpha}^{[1]} = {n_\alpha}^{[0]} - f + 1, \qquad \alpha = h, w,$$

and we define $F * x \in \mathbb{R}^{{n_h}^{[1]} \times {n_w}^{[1]}}$ by

$$(F * x)^k_l = \sum_{i,j=1}^{f^{[1]}} \sum_{\rho=1}^{{n_c}^{[0]}} F^i{}_j{}^\rho x^{i+k-1}{}_{j+l-1}{}^\rho.$$

Similarly, if $p \in \mathbb{Z}_{\geq 0}$ is the padding and $s \in \mathbb{N}$ is the stride, we have that

$${n_\alpha}^{[1]} = \left\lfloor \frac{{n_\alpha}^{[0]} + 2p - f^{[1]}}{s} + 1 \right\rfloor, \qquad \alpha = h, w,$$

and we define $F *_s^p x \in \mathbb{R}^{{n_h}^{[1]} \times {n_w}^{[1]}}$ by

$$(F *_s^p x)^k_l = \sum_{\rho=1}^{{n_c}^{[0]}} \sum_{i,j=1}^{f^{[1]}} F^i{}_j{}^\rho x^{i+s(k-1)-p}{}_{j+s(l-1)-p}{}^\rho \chi_{\mathcal{I}^{[1]}{}^k_l}(i, j).$$

## 11.6 Multiple Filters

Suppose $x \in \mathbb{R}^{{n_h}^{[0]} \times {n_w}^{[0]} \times {n_c}^{[0]}}$, and we wish to convolve $x$ with ${n_c}^{[1]}$-filters, i.e.,

$$F_\eta \in \mathbb{R}^{f^{[1]} \times f^{[1]} \times {n_c}^{[0]}}, \qquad \eta \in \{1, ..., {n_c}^{[1]}\}.$$

Then we have that

$${n_\alpha}^{[1]} = \left\lfloor \frac{{n_\alpha}^{[0]} + 2p - f^{[1]}}{s} + 1 \right\rfloor, \qquad \alpha = h, w,$$

and letting $F = \{F_\eta : 1 \leq \eta \leq {n_c}^{[1]}\}$, we define $F *_s^p x \in \mathbb{R}^{{n_h}^{[1]} \times {n_w}^{[1]} \times {n_c}^{[1]}}$ to be given by

$$(F *_s^p x)_\eta{}^k{}_l = (F_\eta *_s^p x)^k{}_l.$$

# 12 Convolutional Networks

We've now seen enough of how to compute convolutions, and are ready to implement them into a neural network architecture. There are three main types of layers that occur in a convolutional neural network, namely, a convolutional layer (`conv`), a pooling layer (`pool`), and a fully connected layer (`FC`), which the usual type of neural network layer we've seen previously.

## 12.1 Convolutional Layers (conv)

Suppose we are propagating from layer-$\ell$ to layer-$(\ell+1)$ in a neural network, and suppose $a^{[\ell]} \in \mathbb{R}^{{n_h}^{[\ell]} \times {n_w}^{[\ell]} \times {n_c}^{[\ell]}}$. Suppose we have ${n_c}^{[\ell+1]}$-filters we wish to convolve with, each of size $f^{[\ell+1]} \times f^{[\ell+1]} \times {n_c}^{[\ell]}$, and we have padding $p^{[\ell+1]}$ and a stride $s^{[\ell+1]}$. We let $\mathsf{conv}^{[\ell+1]}(a^{[\ell]})$ denote the mapping:

- For $\eta \in \{1, ..., {n_c}^{[\ell]}\}$, compute

$$F_\eta *_{s^{[\ell+1]}}^{p^{[\ell+1]}} a^{[\ell]} + {b_\eta}^{[\ell+1]},$$

  where ${b_\eta}^{[\ell+1]} \in \mathbb{R}$ and the sum is a broadcasting.

- Stack the resultant matrices to obtain an ${n_h}^{[\ell+1]} \times {n_w}^{[\ell+1]} \times {n_c}^{[\ell+1]}$-tensor.

$$\mathsf{conv}^{[\ell+1]}(a^{[\ell]}) = F *_{s^{[\ell+1]}}^{p^{[\ell+1]}} a^{[\ell]} + b^{[\ell+1]}$$

Letting

$$z^{[\ell+1]} = \mathsf{conv}^{[\ell+1]}(a^{[\ell]}),$$

we may then apply our activation unit for the layer $g^{[\ell+1]}$ (broadcasted to the rank-3 tensor). That is, we have $a^{[\ell+1]} \in \mathbb{R}^{{n_h}^{[\ell+1]} \times {n_w}^{[\ell+1]} \times {n_c}^{[\ell+1]}}$ given by

$${a^{[\ell+1]}}_\eta{}^k{}_l = g^{[\ell+1]}({z_\eta}^{[\ell+1]})^k{}_l,$$

where

$${z_\eta}^{[\ell+1]k}{}_l = F_\eta *_{s^{[\ell+1]}}^{p^{[\ell+1]}} a^{[\ell]} + b^{[\ell+1]}.$$

We remark here that the number of parameters we need to train is given by the filters with number of parameters

$$f^{[\ell+1]} \times f^{[\ell+1]} \times {n_c}^{[\ell]} \times {n_c}^{[\ell+1]},$$

plus the bias terms

$$1 \times {n_c}^{[\ell+1]},$$

that is,

$$\begin{aligned}\#(\text{Parameters}) &= f^{[\ell+1]} \times f^{[\ell+1]} \times {n_c}^{[\ell]} \times {n_c}^{[\ell+1]} + 1 \times {n_c}^{[\ell+1]} \\ &= {n_c}^{[\ell+1]} \left({n_c}^{[\ell]}(f^{[\ell+1]})^2 + 1\right)\end{aligned}$$

## 12.2 Pooling Layers (`pool`)

To reduce computational cost and to help prevent over-fitting, a new type of layer is needed to reduce the overall dimensions of the input-size. This is done with a "pooling" layer. There are two main types of pooling layers that we'll discuss here, the *max pooling* layer and the *average pooling* layer.

### 12.2.1 Max Pooling

Suppose

$$x = \begin{bmatrix} 1 & 3 & 2 & 1 \\ 2 & 9 & 1 & 1 \\ 1 & 3 & 2 & 3 \\ 5 & 6 & 1 & 2 \end{bmatrix},$$

and we wish to apply `maxPool` with a "filter size" of $f = 2$, a stride $s = 2$ and padding $p = 0$. Then we apply the max operator to the $(2 \times 2)$-submatrices moving with a stride of 2, i.e., $\texttt{maxPool}(x) \in \mathbb{R}^{2\times 2}$ given by

$$\begin{aligned}\texttt{maxPool}(x) &= \begin{bmatrix} \max\{1,3,2,9\} & \max\{2,1,1,1\} \\ \max\{1,3,5,6\} & \max\{2,3,1,2\} \end{bmatrix} \\ &= \begin{bmatrix} 9 & 2 \\ 6 & 3 \end{bmatrix}.\end{aligned}$$

Since each layer of max pooling has 3 hyper-parameters (and no trainable parameters), we denote these via

$$\texttt{maxPool}_{\{f,p,s\}}(x).$$

### 12.2.2 Average Pooling

Suppose

$$x = \begin{bmatrix} 1 & 3 & 2 & 1 \\ 2 & 9 & 1 & 1 \\ 1 & 3 & 2 & 3 \\ 5 & 6 & 1 & 2 \end{bmatrix},$$

and we wish to apply $\mathtt{avPool}$ with a "filter size" of $f = 2$, a stride of $s = 2$ and padding $p = 0$. Then we apply the averaging operator to the $(2 \times 2)$-submatrices moving with a stride of 2, i.e., $\mathsf{avPool}(x) \in \mathbb{R}^{2\times 2}$ given by

$$\begin{aligned}\mathsf{avPool}(x) &= \begin{bmatrix} \mathbb{E}[\{1,3,2,9\}] & \mathbb{E}[\{2,1,1,1\}] \\ \mathbb{E}[\{1,3,5,6\}] & \mathbb{E}[\{2,3,1,2\}] \end{bmatrix} \\ &= \begin{bmatrix} 3.75 & 1.25 \\ 3.75 & 2 \end{bmatrix}.\end{aligned}$$

Since each layer of average pooling has 3 hyper-parameters (and again, no trainable parameters), we denote these via

$$\mathsf{avPool}_{\{f,p,s\}}(x).$$

## 12.3 A Convolutional Network

Suppose we have a collection of images (our training set), where each image is of the form $x \in \mathbb{R}^{{n_h}^{[0]} \times {n_w}^{[0]} \times {n_c}^{[0]}}$. We shall denote the forward propagation from layer-0 to layer-1 via convolution as the mapping $\mathsf{conv}(1)$ which encompasses the following information:

$$\mathsf{conv}(1) = \begin{cases} \text{filter} \\ \text{padding} \\ \text{stride} \\ \text{number of filter.} \end{cases} .$$

We similarly use $\mathtt{pool}(1)$ to encompass the following information:

$$\mathtt{pool}(1) = \begin{cases} \text{pool type} \\ \text{filter} \\ \text{padding} \\ \text{stride.} \end{cases}$$

This yields a network architecture of the following form:

$$[x] \xrightarrow{\mathsf{conv}^{[1]}} [z^{[1]}] \xrightarrow{\mathtt{pool}^{[1]}} [\zeta^{[1]}] \xrightarrow{g^{[1]}} [a^{[1]}] \xrightarrow{\mathsf{conv}^{[2]}} [z^{[2]}] \xrightarrow{\mathtt{pool}^{[2]}} [\zeta^{[2]}] \xrightarrow{g^{[2]}}$$
$$\xrightarrow{g^{[2]}} [a^{[2]}] \xrightarrow{\text{flatten}} [a^{[2]}] \xrightarrow{\varphi^{[1]}} [z^{[3]}] \xrightarrow{g^{[3]}} [a^{[3]}] \longrightarrow \cdots \longrightarrow \hat{y}$$

We remark here that the convolution and pooling layers are done before the fully connected layers. Moreover, we apply the nonlinearity after the pooling, but this doesn't matter when doing max pooling, since our nonlinearities are typically non-decreasing. We choose this order because it's typically computationally cheaper.

We also remark that since each output of a convolutional layer only depends on a subset of features, our model is less prone to over-fitting.

## 12.4 Backpropagation

We introduce the following tensoral notation: We say $x \in \mathbb{R}^{a}{}_{b,c}$ is a $(1,2)$-tensor written in index form

$$x = (x^{\rho}{}_{ij})$$

with $1 \leq \rho \leq a$, $1 \leq i \leq b$ and $1 \leq j \leq c$. Similarly, we say $W \in \mathbb{R}^{a,b,c}{}_{d}$ is a $(3,1)$-tensor written in index form

$$W = (W^{\eta ij}{}_{\rho}).$$

Suppose $x \in \mathbb{R}^{n_c}{}_{n_h,n_w}$, $W \in \mathbb{R}^{m_c,f,f}{}_{n_c}$ and $b \in \mathbb{R}^{m_c}$ with padding $p \geq 0$ and stride $s \in \mathbb{N}$. Then we have that

$$z = \mathsf{conv}(x) \in \mathbb{R}^{m_c}{}_{m_h,m_w}$$

is given by

$$z^{\eta}{}_{k,l} = \sum_{\rho=1}^{n_c} \sum_{i,j=1}^{f} W^{\eta,i,j}{}_{\rho} x^{\rho}{}_{i+s(k-1)-p,j+s(l-1)-p} \chi_{\mathcal{I}_{k,l}}(i,j) + b^{\eta}.$$

This is the general formula for the forward propagation of a $\mathsf{conv}$ layer.

We now compute derivatives for general loss function $\mathbb{L}$:

$$\frac{\partial z^{\eta}{}_{k,l}}{\partial b^{\mu}} = \delta^{\eta}_{\mu},$$

and hence

$$\begin{aligned}\frac{\partial \mathbb{L}}{\partial b^{\mu}} &= \sum_{\eta=1}^{m_c} \sum_{k=1}^{m_h} \sum_{l=1}^{m_w} \frac{\partial \mathbb{L}}{\partial z^{\eta}{}_{k,l}} \frac{\partial z^{\eta}{}_{k,l}}{\partial b^{\mu}} \\ &= \sum_{\eta=1}^{m_c} \sum_{k=1}^{m_h} \sum_{l=1}^{m_w} \frac{\partial \mathbb{L}}{\partial z^{\eta}{}_{k,l}} \delta^{\eta}_{\mu} \\ &= \sum_{k=1}^{m_h} \sum_{l=1}^{m_w} \frac{\partial \mathbb{L}}{\partial z^{\mu}{}_{k,l}}.\end{aligned}$$

Next we consider

$$\begin{aligned}\frac{\partial {z^\eta}_{k,l}}{\partial {W^{\alpha,\mu,\nu}}_\beta} &= \sum_{\rho=1}^{n_c}\sum_{i,j=1}^{f} \delta^\eta_\alpha \delta^i_\mu \delta^j_\nu \delta^\beta_\rho {x^\rho}_{i+s(k-1)-p,j+s(l-1)-p}\chi_{\mathcal{I}_{k,l}}(i,j)\\ &= \delta^\eta_\alpha {x^\beta}_{\mu+s(k-1)-p,\nu+s(l-1)-p}\chi_{\mathcal{I}_{k,l}}(\mu,\nu)\end{aligned}$$

and hence

$$\begin{aligned}\frac{\partial \mathbb{L}}{\partial {W^{\alpha,\mu,\nu}}_\beta} &= \sum_{\eta=1}^{m_c}\sum_{k=1}^{m_h}\sum_{l=1}^{m_w} \frac{\partial \mathbb{L}}{\partial {z^\eta}_{k,l}}\frac{\partial {z^\eta}_{k,l}}{\partial {W^{\alpha,\mu,\nu}}_\beta}\\ &= \sum_{\eta=1}^{m_c}\sum_{k=1}^{m_h}\sum_{l=1}^{m_w} \frac{\partial \mathbb{L}}{\partial {z^\eta}_{k,l}}\delta^\eta_\alpha {x^\beta}_{\mu+s(k-1)-p,\nu+s(l-1)-p}\chi_{\mathcal{I}_{k,l}}(\mu,\nu)\\ &= \sum_{k=1}^{m_h}\sum_{l=1}^{m_w} \frac{\partial \mathbb{L}}{\partial {z^\alpha}_{k,l}} {x^\beta}_{\mu+s(k-1)-p,\nu+s(l-1)-p}\chi_{\mathcal{I}_{k,l}}(\mu,\nu).\end{aligned}$$

Finally, we consider

$$\begin{aligned}\frac{\partial {z^\eta}_{k,l}}{\partial {x^\alpha}_{\mu,\nu}} &= \sum_{\rho=1}^{n_c}\sum_{i,j=1}^{f} {W^{\eta,i,j}}_\rho \delta^\rho_\alpha \delta^\mu_{i+s(k-1)-p} \delta^\nu_{j+s(l-1)-p}\chi_{\mathcal{I}_{k,l}}(i,j)\\ &= {W^{\eta,\mu-p-s(k-1),\nu-p-s(l-1)}}_\alpha \chi_{\mathcal{I}_{k,l}}(\mu-p-s(k-1),\nu-p-s(l-1))\\ &= {W^{\eta,\mu-p-s(k-1),\nu-p-s(l-1)}}_\alpha \begin{cases}1 & \text{if } p<(\mu,\nu)\leq p+(n_h,n_w)\\ 0 & \text{else}\end{cases},\end{aligned}$$

and hence

$$\begin{aligned}\frac{\partial \mathbb{L}}{\partial {x^\alpha}_\mu, \nu} &= \sum_{\eta=1}^{m_c}\sum_{k=1}^{m_h}\sum_{l=1}^{m_w} \frac{\partial \mathbb{L}}{\partial {z^\eta}_{k,l}}\frac{\partial {z^\eta}_{k,l}}{\partial {x^\alpha}_{\mu,\nu}}\\ &=\end{aligned}$$

# Appendices

## A utils.py

```
#! python3
import copy

import numpy as np
from sklearn.utils import shuffle

import mlLib.npActivators as npActivators
from mlLib.npActivators import ACTIVATORS

## Usefule printing function
def print_array_dict(D):
    """
    Parameters
    ----------
    D : Dict[array_like]

    Returns
    -------
    None
    """
    txt = "Array␣{0}␣has␣shape␣{1}\n{2}"
    for k, v in D.items():
        print(txt.format(str(k), v.shape, v))


## Partition data into training, development, and test sets
def partition_data(x, y, train_ratio):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (m, N)
    y : array_like
        y.shape = (k, N)
    train_ratio : float
        0<=train_ratio<=1

    Returns
    -------
    train : Tuple[array_like]
    dev : Tuple[array_like]
    test : Tuple[array_like]
    """
    ## Shuffle the data
    x, y = shuffle(x.T, y.T) # Only shuffles rows, so transpose is needed
    x = x.T
    y = y.T

    ## Get the size of partitions
    N = x.shape[1]
    N_train = int(train_ratio * N)
    N_mid = (N - N_train) // 2

    ## Create partitions
    train = (x[:,:N_train], y[:,:N_train])
    dev = (x[:,N_train:N_train + N_mid], y[:,N_train:N_train + N_mid])
    test = (x[:,N_train + N_mid:], y[:,N_train + N_mid:])

    assert(x.all() == np.concatenate([train[0], dev[0], test[0]], axis=1).all())
    assert(y.all() == np.concatenate([train[1], dev[1], test[1]], axis=1).all())

    return train, dev, test

## Partition training data into batches
def get_batches(x, y, b):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (m, n)
    y : array_like
        y.shape = (k, n)
    b : int

    Returns
    -------
    batches : List[Dict]
        batches[i][’x’] : array_like
            x.shape = (m, b) # except last batch
            y.shape = (k, b) # except last batch

    """
    m, n = x.shape
    B = int(np.ceil(n / b))
    batches = []
    for i in range(B):
        x_temp = x[:,(b * i):(b * (i + 1))]
        y_temp = y[:,(b * i):(b * (i + 1))]
        batches.append({’x’ : x_temp, ’y’ : y_temp})
    # Slicing automatically ends at the end of
    # the list if the stop is outside the index
    return batches

##### General Neural Network Model #####

## Retrieve number of examples and layer dimensions
def dim_retrieval(x, y, hidden_sizes):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[L], n)
    hidden_sizes : List[int]
        hidden_sizes[i-1] = The number nodes layer i
    Returns
    -------
    n : int
        The number of training examples
    layers : List
        layer[l] = # nodes in layer l

    """
    m, n = x.shape
    assert(y.shape[1] == n)
    K = y.shape[0]
    layers = [m]
    layers.extend(hidden_sizes)
    layers.append(K)

    return n, layers

## Initialize parameters using the size of each layer
def initialize_parameters_random(layers):
    """
    Parameters
    ----------
    layers : List[int]
        layers[l] = # nodes in layer l
    Returns
    -------
    params : Dict[Dict]
        w[l] : array_like
            dwl.shape = (layers[l], layers[l-1])
        b[l] : array_like
            dbl.shape = (layers[l], 1)
    """
    w = {}
    b = {}
    for l in range(1, len(layers)):
        w[l] = np.random.randn(layers[l], layers[l - 1]) * 0.01
        b[l] = np.zeros((layers[l], 1))
    params = {’w’ : w, ’b’ : b}
    return params

## Forward and Backward Linear Activations
def linear_activation_forward(a_prev, w, b, activator):
    """
    Parameters
    ----------
    a_prev : array_like
        a_prev.shape = (layers[l], n)
    w : array_like
        w.shape = (layers[l+1], layers[l])
    b : array_like
        b.shape = (layers[l+1], 1)
    activator : str
        activator in ACTIVATORS

    Returns
    -------
    z : array_like
        z.shape = (layer_dims[l+1], n)
    a : array_like
        a.shape = (layer_dims[l+1], n)
    """
    assert activator in ACTIVATORS, f’{activator}␣is␣not␣a␣valid␣activator.’

    z = w @ a_prev + b
    if activator == ’relu’:
        a, _ = npActivators.relu(z)
    elif activator == ’sigmoid’:
        a, _ = npActivators.sigmoid(z)
    elif activator == ’tanh’:
        a, _ = npActivators.tanh(z)
    return z, a

def linear_activation_backward(delta_next, z, w, activator):
    """
    Parameters
    ----------
    delta_next : array_like
        delta_next.shape = (layers[l+1], n)
    z : array_like
        z.shape = (layers[l+1], n)
    w : array_like
        w.shape = (layers[l+1], layers[l])
    activator : str
        activator in ACTIVATORS

    Returns
    -------
    delta : array_like
        delta.shape = (layers[l], n)
    """
    assert activator in ACTIVATORS, f’{activator}␣is␣not␣a␣valid␣activator.’

    n = delta_next.shape[1]

    if activator == ’relu’:
        _, dg = npActivators.relu(z)
    elif activator == ’sigmoid’:
        _, dg = npActivators.sigmoid(z)
    elif activator == ’tanh’:
        _, dg = npActivators.tanh(z)

    da = w.T @ delta_next
    assert(da.shape == (w.shape[1], n))
    delta = da * dg
    assert(delta.shape == (w.shape[1], n))
    return delta


## Forward and Backward Propagation with Dropout Regularization
# Generate dropout matrices
def dropout_matrices(layers, num_examples, keep_prob):
    """
    Parameters
    ----------
    layers : List[int]
        layers[l] = number of nodes in layer l
    num_examples : int
        The number of training examples
    keep_prob : List[float]
        keep_prob[l] = The probabilty of keeping a node in layer l

    Returns
    -------
    D : Dict[array_like]
        D[l].shape = (layers[l], num_ex)
        D[l] = a Boolean array
    """
    np.random.seed(1)
    L = len(layers)
    D = {}
    for l in range(L - 1):
        D[l] = np.random.rand(layers[l], num_examples)
        D[l] = (D[l] < keep_prob[l]).astype(int)
        assert(D[l].shape == (layers[l], num_examples))
    return D

def forward_propagation_dropout(x, params, activators, D, keep_prob):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0] n)
    params : Dict[Dict]
        params['w'][l] : array_like
            wl.shape = (layers[l], layers[l-1])
        params['b'][l] : array_like
            bl.shape = (layers[l], 1)
    activators : List[str]
        activators[l] = activation function of layer l+1
    D : Dict[array_like]
        D[l].shape = (layer_dims[l], num_ex)
        D[l] = a Boolean array astype(int)
    keep_prob : List[float]
        keep_prob[l] = The probabilty of keeping a node in layer l

    Returns
    -------
    cache : Dict[Dict]
        cache['z'][l] : array_like
            z[l].shape = (layers[l], n)
        cache['a'][l] : array_like
            a[l].shape = (layers[l], n)
    """
    # Retrieve parameters
    w = params['w']
    b = params['b']
    L = len(w) # Number of layers excluding output layer
    n = x.shape[1]
    # Set empty caches
    a = {}
    z = {}
    # Dropout on layer 0
    a[0] = x
    a[0] = a[0] * D[0]
    a[0] /= keep_prob[0]
    # Loop through hidden layers
    for l in range(1, L + 1):
        zl, al = linear_activation_forward(a[l - 1], w[l], b[l], activators[l - 1])
        al = al * D[l]
        al /= keep_prob[l]
        z[l] = zl
        a[l] = al
    # Output layer
    z[L], a[L] = linear_activation_forward(a[L - 1], w[L], b[L], activators[-1])

    cache = {’z’ : z, ’a’ : a}
    return cache

def backward_propagation_dropout(x, y, params, cache, activators, D, keep_prob):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[-1], n)
    params : Dict[Dict[array_like]]
        params[’w’][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            b[l].shape = (layers[l], 1)
    cache : Dict[Dict[array_like]]
        cache[’a’][l] : array_like
            a[l].shape = (layers[l], n)
        cache[’z’][l] : array_like
            z[l].shape = (layers[l], n)
    activators : List[str]
        activators[l] = activation function of layer l+1
    D : Dict[array_like]
        D[l].shape = (layer_dims[l], num_ex)
        D[l] = a Boolean array astype(int)
    keep_prob : List[float]
        keep_prob[l] = The probabilty of keeping a node in layer l

    Returns
    -------
    grads : Dict[Dict]
```

```
        grads['dw'][l] : array_like
            dw[l].shape = w[l].shape
        grads['db'][l] : array_like
            db[l].shape = b[l].shape
    """
    ## Retrieve parameters
    a = cache['a']
    z = cache['z']
    w = params['w']
    n = x.shape[1]
    L = len(z)

    ## Compute deltas
    delta = {}
    delta[L] = a[L] - y
    for l in reversed(range(1, L)):
        deltal = linear_activation_backward(delta[l + 1], z[l], w[l + 1], activators
        deltal = deltal * D[l]
        deltal /= keep_prob[l]
        delta[l] = deltal

    ## Compute gradients
    dw = {}
    db = {}

    for l in range(1, L + 1):
        db[l] = (1 / n) * np.sum(delta[l], axis=1, keepdims=True)
        assert(db[l].shape == (w[l].shape[0], 1))
        dw[l] = (1 / n) * delta[l] @ a[l - 1].T
        assert(dw[l].shape == w[l].shape)
    grads = {'w' : dw, 'b' : db}
    return grads


## Forward and Backward Propagation with L2-Regularization
def forward_propagation(x, params, activators):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0] n)
    params : Dict[Dict]
        params['w'][l] : array_like
            wl.shape = (layers[l], layers[l-1])
        params['b'][l] : array_like
            bl.shape = (layers[l], 1)
    activators : List[str]
```

```python
        activators[l] = activation function of layer l+1
    Returns
    -------
    cache : Dict[Dict]
        cache['z'][l] : array_like
            z[l].shape = (layers[l], n)
        cache['a'][l] : array_like
            a[l].shape = (layers[l], n)
    """
    # Retrieve parameters
    w = params['w']
    b = params['b']
    L = len(w) # Number of layers excluding output layer
    n = x.shape[1]
    # Set empty caches
    a = {}
    z = {}
    # Initialize a
    a[0] = x
    for l in range(1, L + 1):
        z[l], a[l] = linear_activation_forward(a[l - 1], w[l], b[l], activators[l -

    cache = {'a' : a, 'z' : z}
    return cache

def backward_propagation(x, y, params, cache, activators, lambda_=0.0):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[-1], n)
    params : Dict[Dict[array_like]]
        params['w'][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params['b'][l] : array_like
            b[l].shape = (layers[l], 1)
    cache : Dict[Dict[array_like]]
        cache['a'][l] : array_like
            a[l].shape = (layers[l], n)
        cache['z'][l] : array_like
            z[l].shape = (layers[l], n)
    activators : List[str]
        activators[l] = activation function of layer l+1
    lambda_ : float
        Default: 0.0

    Returns
    -------
    grads : Dict[Dict]
        grads[’w’][l] : array_like
            dw[l].shape = w[l].shape
        grads[’b’][l] : array_like
            db[l].shape = b[l].shape
    """
    ## Retrieve parameters
    a = cache[’a’]
    z = cache[’z’]
    w = params[’w’]
    n = x.shape[1]
    L = len(z)

    ## Compute deltas
    delta = {}
    delta[L] = a[L] - y
    for l in reversed(range(1, L)):
        delta[l] = linear_activation_backward(delta[l + 1], z[l], w[l + 1], activato

    ## Compute gradients
    dw = {}
    db = {}
    for l in range(1, L + 1):
        db[l] = (1 / n) * np.sum(delta[l], axis=1, keepdims=True)
        assert(db[l].shape == (w[l].shape[0], 1))
        dw[l] = (1 / n) * (delta[l] @ a[l - 1].T + lambda_ * w[l])
        assert(dw[l].shape == w[l].shape)
    grads ={’w’ : dw, ’b’ : db}
    return grads


## Compute the (L2-regulated) cost
def compute_cost(y, params, cache, lambda_=0.0):
    """
    Parameters
    ----------
    y : array_like
        y.shape = (layers[-1], n)
    params : Dict[Dict[array_like]]
        params[’w’][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            b[l].shape = (layers[l], 1)
    cache : Dict[Dict[array_like]]
```

```
        cache[’z’][l] : array_like
            z[l].shape = (layers[l], n)
        cache[’a’][l] : array_like
            a[l].shape = (layers[l], n)
    lambda_ : float
        Default: 0.0

    Returns
    -------
    cost : float
        The cost evaluated at y and aL
    """
    ## Retrieve parameters
    n = y.shape[1]
    a = cache[’a’]
    w = params[’w’]
    L = len(a)
    aL = a[L - 1]

    ## Regularization term
    R = 0
    for l in range(1, L):
        R += np.sum(w[l] * w[l])
    R *= (lambda_ / (2 * n))

    ## Unregularized cost
    J = (-1 / n) * (np.sum(y * np.log(aL)) + np.sum((1 - y) * np.log(1 - aL)))

    ## Total Cost
    cost = J + R
    cost = float(np.squeeze(cost))
    return cost


## Update parameters via gradient descent
def update_parameters(params, grads, learning_rate=0.01):
    """
    Parameters
    ----------
    params : Dict[Dict]
        params[’w’][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            b[l].shape = (layers[l], 1)
    grads : Dict[Dict]
        grads[’w’][l] : array_like
            dw[l].shape = w[l].shape
        grads['b'][l] : array_like
            db[l].shape = b[l].shape
    learning_rate : float
        Default: 0.01
        The learning rate for gradient descent

    Returns
    -------
    params : Dict[Dict]
        params['w'][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params['b'][l] : array_like
            b[l].shape = (layers[l], 1)
    """
    ## Retrieve parameters
    w = copy.deepcopy(params['w'])
    b = copy.deepcopy(params['b'])
    L = len(w)

    ## Retrieve gradients
    dw = grads['w']
    db = grads['b']

    ## Perform update
    for l in range(1, L + 1):
        w[l] = w[l] - learning_rate * dw[l]
        b[l] = b[l] - learning_rate * db[l]

    params = {'w' : w, 'b' : b}
    return params










#### Dropout NN Model ####
def model_nn(x, y, hidden_layer_sizes, activators, keep_prob=1.0, num_iters=10000,
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
```

```
        y.shape = (layers[-1], n)
    hidden_layer_sizes : List[int]
        The number nodes layer l = hidden_layer_sizes[l-1]
    activators : List[str]
        activators[l] = activation function of layer l+1
    keep_prob : List[float] | float
        keep_prob[l] = The probabilty of keeping a node in layer l
        keep_prob = The same probability for all input and hidden layers
    num_iters : int
        Number of iterations with which our model performs gradient descent
    print_cost : Boolean
        If True, print the cost every 1000 iterations

    Returns
    -------
    params : Dict[Dict]
        params[’w’][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            b[l].shape = (layers[l], 1)
    cost : float
        The final cost value for the optimized parameters returned
    """
    ## Set dimensions and Initialize parameters
    n, layers = dim_retrieval(x, y, hidden_layer_sizes)
    params = initialize_parameters_random(layers)

    ## Expand keep_prob to a list if it’s a single float
    if isinstance(keep_prob, float):
        keep_prob = [keep_prob] * (len(layers) - 1)

    # main gradient descent loop
    for i in range(num_iters):
        D = dropout_matrices(layers, n, keep_prob)
        cache = forward_propagation(x, params, activators, D, keep_prob)
        cost = compute_cost(cache, y)
        grads = backward_propagation(x, y, params, cache, activators, D, keep_prob)
        params = update_parameters(params, grads)

        if print_cost and i % 1000 == 0:
            print(f’Cost␣after␣iteration␣{i}:␣{cost}’)

    return params, cost
```

```
########## TESTING ##########
def test_dropout_nn():
    x = np.random.rand(4, 500)
    y = np.random.rand(1, 500)
    hidden_layer_sizes = [4, 5, 4]
    activators = [’relu’, ’relu’, ’relu’, ’sigmoid’]
    keep_prob = 1.0
    params, cost = model_nn(x, y, hidden_layer_sizes, activators, keep_prob)
    print(params)



######## Functions to use later
def reshape_labels(num_labels, y):
    ”””
    Parameters
    ----------
    num_labels : int
        The number of possible labels the output y may take
    y : array_like
        y.size = n
        y[i] takes values in {1,2,...,num_labels}
    Returns
    Y : array_like
        Y.shape = (num_lables, n)
        Y[i][j] = 1 if y[j] = i, Y[i][j] = 0 otherwise
    -------
    ”””

    if num_labels <= 2:
        return y
    else:
        omega = []
        for i in range(num_labels):
            omega.append(np.eye(1, num_labels, i))  # the standard i-th basis vector

        Y = np.concatenate([omega[i] for i in y], axis=0).T
        for i in range(num_labels):
            for j in range(n):
                if y[j] == i:
                    assert Y[i][j] == 1
                else:
                    assert Y[i][j] == 0
        return Y
```

```python

#######
if __name__ == '__main__':
    test_dropout_nn()
```

# B activators.py

```python
import numpy as np

ACTIVATORS = ['relu', 'sigmoid', 'tanh', 'linear', 'softmax']

## Activator functions
# The (leaky-)ReLU function
def relu(z, beta=0.0):
    """
    Parameters
    ----------
    z : array_like
    beta : float

    Returns
    -------
    r : array_like
        The (broadcasted) ReLU function when beta=0, the leaky-ReLU otherwise.
    dr : array_like
        The (broadcasted) derivative of the (leaky-)ReLU function
    """
    # Change scalar to array if needed
    z = np.array(z)
    # Compute value of ReLU(z)
    r = np.maximum(z, beta * z)
    # Compute differential ReLU'(z)
    dr = ((~(z < 0)) * 1) + ((z < 0) * beta)
    return r, dr

# The sigmoid function
def sigmoid(z):
    """
    Parameters
    ----------
    z : array_like

    Returns
    -------
    sigma : array_like
        The (broadcasted) value of the sigmoid function evaluated at z
    dsigma : array_like
        The (broadcasted) derivative of the sigmoid function evaluate at z
    """
    # Compute value of sigmoid
    sigma = (1 / (1 + np.exp(-z)))
    # Compute differential of sigmoid
    dsigma = sigma * (1 - sigma)
    return sigma, dsigma

# The hyperbolic tangent function
def tanh(z):
    """
    Parameters
    ----------
    z : array_like

    Returns
    phi : array_like
        The (broadcasted) value of the hyperbolic tangent function evaluated at z
    dphi : array_like
        The (broadcasted) derivative of hyperbolic tangent function evaluated at z
    """
    # Compute value of tanh
    phi = np.tanh(z)
    # Compute differential of tanh
    dphi = 1 - (phi * phi)
    return phi, dphi

# The linear activator function
def linear(z):
    """
    Parameters
    ----------
    z : array_like

    Returns
    -------
    id : array_like
    d_id
    """
    id = z
    d_id = np.ones(z.shape)
    return id, d_id
```

# C The Normalization Operator

sec:normOp

In this section, we wish to character the (reverse) differential of the normalization operator $N : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$ given in coordinates by

$$N : x^i_j \mapsto \frac{x^i_j - \mathbb{E}[x^i]}{\sqrt{\mathbb{V}[x^i] + \epsilon}}.$$

First, let's rewrite this without coordinates

$$\begin{aligned} N(x) &= (\mathbb{V}[x] + \vec{\epsilon})^{\odot -\frac{1}{2}} \vec{1}^T \odot (x - \mathbb{E}[x]\vec{1}^T) \\ &=: f(x) \odot g(x). \end{aligned}$$

Now, let's fix $(x, \xi) \in T\mathbb{R}^{m\times n}$, and we immediately see that the Hadamard product obeys the Leibniz Rule with exterior differentiation, i.e.,

$$dN_x(\xi) = g(x) \odot df_x(\xi) + f(x) \odot dg_x(\xi),$$

so we consider these computations separately. Moreover, we now need to compute the derivative of the expectation $\mathbb{E}$ and variance $\mathbb{V}$ operators.

1. For the expectation of a random vector, $\mathbb{E} : \mathbb{R}^{m\times n} \to \mathbb{R}^m$, we first rewrite $\mathbb{E}$ as follows

   $$\begin{aligned} \mathbb{E}[x] &= \sum_{i=1}^{m} \left( \frac{1}{n} \sum_{j=1}^{n} x^i_j \right) e_i \\ &= \frac{1}{n} x \cdot \vec{1} \\ &= \frac{1}{n} R_{\vec{1}}(x) \end{aligned}$$

   where $\vec{1} = (1, 1, ..., 1) \in \mathbb{R}^n$. This is clearly linear, so for $(x, \xi) \in T\mathbb{R}^{m\times n}$, we have that

   $$\begin{aligned} d\mathbb{E}_x(\xi) &= \mathbb{E}[\xi] \\ &= \frac{1}{n} R_{\vec{1}}(\xi). \end{aligned}$$

   For a fixed $x \in \mathbb{R}^{m\times n}$, we let $\mu := \mathbb{E}[x] \in \mathbb{R}^m$ denote the output.

2. For the variance of a random vector, $\mathbb{V} : \mathbb{R}^{m\times n} \to \mathbb{R}^m$, we rewrite $\mathbb{V}$ as follows

$$\begin{aligned} \mathbb{V}[x] &= \sum_{i=1}^{m} \left( \frac{1}{n} \sum_{j=1}^{n} (x_j^i - \mu^i)^2 \right) e_i \\ &= \mathbb{E}\left[ (x - \mu\vec{1}^T) \odot (x - \mu\vec{1}^T) \right] \\ &= \mathbb{E}\left[ (x - \mu\vec{1}^T)^{\odot 2} \right]. \end{aligned}$$

From the first calculation, we know how to compute the derivative of $\mathbb{E}$, so we focus on the input $(x - \mu\vec{1}^T)^{\odot 2}$.

To this end, we define $\psi : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$ to be the inner-most function given by

$$\begin{aligned} \psi(x) &= x - \mathbb{E}[x]\vec{1}^T \\ &= \left( \mathbb{1}_{\mathbb{R}^{m\times n}} - R_{\vec{1}^T} \circ \mathbb{E} \right)(x), \end{aligned}$$

which is clearly linear. Then for $(x, \xi) \in T\mathbb{R}^{m\times n}$ we see that

$$d\psi_x(\xi) = \left( \mathbb{1}_{T_x\mathbb{R}^{m\times n}} - \frac{1}{n} R_{\vec{1}^T} \circ R_{\vec{1}} \right)(\xi),$$

where we used our previous computation for $d\mathbb{E}_x$.

Next, define $\phi : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$ to be the Hadamard-square, i.e.,

$$\phi(x) = x^{\odot 2} = x \odot x.$$

Using our previous remark of the Leibniz Rule in regard to the Hadamard product, we see that for $(x, \xi) \in T\mathbb{R}^{m\times n}$,

$$\begin{aligned} d\phi_x(\xi) &= x \odot \xi + \xi \odot x \\ &= 2x \odot \xi \\ &= \odot_{2x}(\xi). \end{aligned}$$

Finally, by the compositional definition of $\mathbb{V}$,

$$\mathbb{V}[x] = \mathbb{E} \circ \phi \circ \psi(x),$$

we compute for any $(x, \xi) \in T\mathbb{R}^{m \times n}$ that

$$\begin{aligned}
d\mathbb{V}_x(\xi) &= d\mathbb{E}_{\phi(\psi(x))} \circ d\phi_{\psi(x)} \circ d\psi_x(\xi) \\
&= d\mathbb{E}_{\phi(\psi(x))} \circ d\phi_{\psi(x)} \left( \xi - \frac{1}{n}\xi\vec{1}\vec{1}^T \right) \\
&= d\mathbb{E}_{\phi(\psi(x))} \left( 2(x - \mu\vec{1}^T) \odot \left( \xi - \frac{1}{n}\xi\vec{1}\vec{1}^T \right) \right) \\
&= \mathbb{E}\left[ 2(x - \mu\vec{1}^T) \odot \left( \xi - \frac{1}{n}\xi\vec{1}\vec{1}^T \right) \right] \\
&= \mathbb{E}\left[ 2(x - \mu\vec{1}^T) \odot \xi \right] - 2\mathbb{E}\left[ (x - \mu\vec{1}^T) \odot (\mathbb{E}[\xi]\vec{1}^T) \right].
\end{aligned}$$

Next, we notice that if we let $\gamma := \mathbb{E}[\xi] \in \mathbb{R}^m$, then

$$\begin{aligned}
\gamma\vec{1}^T &= \begin{pmatrix} \gamma^1 \\ \vdots \\ \gamma^m \end{pmatrix} \begin{pmatrix} 1 & 1 & \cdots & 1 \end{pmatrix} \\
&= \begin{bmatrix} \gamma^1 & \gamma^1 & \cdots & \gamma^1 \\ \gamma^2 & \gamma^2 & \cdots & \gamma^2 \\ \vdots & \vdots & \ddots & \vdots \\ \gamma^m & \gamma^m & \cdots & \gamma^m \end{bmatrix} \in \mathbb{R}^{m \times n}
\end{aligned}$$

and hence that

$$\begin{aligned}
\mathbb{E}[(x - \mu\vec{1}^T) \odot \gamma\vec{1}^T] &= \sum_{i=1}^{m} \left( \frac{1}{n} \sum_{j=1}^{n} (x_j^i - \mu^i)\gamma^i \right) e_i \\
&= \sum_{i=1}^{m} \left( \gamma^i(\mathbb{E}[x^i] - \mu^i) \right) e_i \\
&= 0.
\end{aligned}$$

Resuming our computation, we now have that

$$\begin{aligned}
d\mathbb{V}_x(\xi) &= \mathbb{E}\left[ 2(x - \mu\vec{1}^T) \odot \xi \right] \\
&= \frac{2}{n} R_{\vec{1}} \circ \odot_{x - \mu\vec{1}^T}(\xi)
\end{aligned}$$

We remark that for a fixed $x \in \mathbb{R}^{m \times n}$, we let $\sigma^2 := \mathbb{V}[x]$ denote the output.

We have now computed the following differentials for any $(x, \xi) \in T\mathbb{R}^{m\times n}$,

$$\begin{aligned} d\mathbb{E}_x(\xi) &= \frac{1}{n} R_{\vec{1}}(\xi), \\ d\mathbb{V}_x(\xi) &= \frac{2}{n} R_{\vec{1}} \circ \odot_{x-\mu\vec{1}^T}(\xi), \end{aligned}$$

and are now ready to compute the differentials of our previously defined $f$ and $g$, that is,

$$f : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}, \qquad f(x) = (\mathbb{V}[x] + \vec{\epsilon})^{\odot -\frac{1}{2}}\, \vec{1}^T,$$

and

$$g : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}, \qquad g(x) = x - \mathbb{E}[x]\vec{1}^T.$$

However, we see here that $g \equiv \psi$ as defined, and so for any $(x, \xi) \in T\mathbb{R}^{m\times n}$, we have that

$$dg_x(\xi) = \left(\mathbb{1} - \frac{1}{n} R_{\vec{1}^T} \circ R_{\vec{1}}\right)(\xi) = \xi - \frac{1}{n}\xi\vec{1}\vec{1}^T.$$

Hence we need only focus on $f$. To this end, for $(x, \xi) \in T\mathbb{R}^{m\times n}$, we first compute the differential of the Hadamard-root operator, $h(x) = x^{\odot -\frac{1}{2}}$,

$$\begin{aligned} dh_x(\xi) &= \left.\frac{d}{dt}\right|_{t=0} (x + t\xi)^{\odot -\frac{1}{2}} \\ &= \left.\frac{d}{dt}\right|_{t=0} \left[(x^i_j + t\xi^i_j)^{-\frac{1}{2}}\right] \\ &= \left[-\tfrac{1}{2}(x^i_j)^{-\frac{3}{2}}\xi^i_j\right] \\ &= -\frac{1}{2} x^{\odot -\frac{3}{2}} \odot \xi \\ &= -\frac{1}{2} \odot_{x^{\odot -\frac{3}{2}}} (\xi). \end{aligned}$$

After writing $f$ as the composition

$$f(x) = R_{\vec{1}^T} \circ h(\mathbb{V}[x] + \vec{\epsilon}),$$

we now compute

$$\begin{aligned} df_x(\xi) &= R_{\vec{1}^T} \circ dh_{\sigma^2+\vec{\epsilon}} \circ d\mathbb{V}_x(\xi) \\ &= -\frac{1}{n} R_{\vec{1}^T} \circ \odot_{(\sigma^2+\vec{\epsilon})^{\odot -\frac{3}{2}}} \circ R_{\vec{1}} \circ \odot_{x-\mu\vec{1}^T}(\xi) \\ &= -\frac{1}{n}(\sigma^2 + \vec{\epsilon})^{\odot -\frac{3}{2}}\vec{1}^T \odot (x - \mu\vec{1}^T) \odot \xi\vec{1} \end{aligned}$$

Finally, recalling that we defined

$$N(x) = f(x) \odot g(x),$$

and so we have that

$$\begin{aligned} dN_x(\xi) &= g(x) \odot df_x(\xi) + f(x) \odot dg_x(\xi) \\ &= -\frac{1}{n} \odot_{x-\mu\vec{1}^T} R_{\vec{1}^T} \circ \odot_{(\sigma^2+\vec{\epsilon})^{\odot-\frac{3}{2}}} \circ R_{\vec{1}} \circ \odot_{x-\mu\vec{1}^T}(\xi) \\ &\qquad + (\sigma^2 + \vec{\epsilon})^{\odot-\frac{1}{2}} \vec{1}^T \odot \left(\mathbb{1} - \frac{1}{n} R_{\vec{1}^T} \circ R_{\vec{1}}\right)(\xi). \end{aligned}$$

To simplify the expression for implementation in python, we make the auxiliary definitions (which only depend on the forward propagating computations)

$$y := N(x),$$

and

$$\theta := (\sigma^2 + \vec{\epsilon})^{\odot-\frac{1}{2}}.$$

Then our computation reduces to

$$dN_x(\xi) = -\frac{1}{n}(x - \mu\vec{1}^T)\odot$$

$$\begin{aligned} dN_x(\xi) &= -\frac{1}{n}\eta \circ R_{\vec{1}^T} \circ \Theta \circ R_{\vec{1}} \circ \eta(\xi) + \theta \circ \left(\mathbb{1} - \frac{1}{n} R_{\vec{1}^T} \circ R_{\vec{1}}\right)(\xi) \\ &= \left[-\frac{1}{n}\eta \circ R_{\vec{1}^T} \circ \Theta \circ R_{\vec{1}} \circ \eta + \theta - \frac{1}{n}\theta \circ R_{\vec{1}^T} \circ R_{\vec{1}}\right](\xi). \end{aligned}$$

Then for $\zeta \in T_{N(x)}\mathbb{R}^{m \times n}$, we have the reverse differential

$$\begin{aligned} \langle rN_x(\zeta), \xi\rangle_F &= \langle \zeta, dN_x(\xi)\rangle_F \\ &= \left\langle \left[-\frac{1}{n}\eta \circ R_{\vec{1}^T} \circ \Theta \circ R_{\vec{1}} \circ \eta + \theta - \frac{1}{n} R_{\vec{1}^T} \circ R_{\vec{1}} \circ \theta\right](\zeta), \xi \right\rangle_F . \end{aligned}$$

## C.1 The Normalization Operator v.2

Suppose $N : (\mathbb{R}^m)^n \to (\mathbb{R}^m)^n$ is given by

$$N(x_1, ..., x_n) = (y_1, ..., y_n),$$

where
$$y_j = \frac{x_j - \mathbb{E}[x]}{\sqrt{\mathbb{V}[x] + \epsilon}}.$$
Then for $(x, \xi) \in T(\mathbb{R}^m)^n$, we have that
$$dN_x(\xi) = \bigoplus_{j=1}^{n} d_j N_x(\xi_j).$$

For what follows, we fix $x \in (\mathbb{R}^m)^n$, $\alpha, \beta \in \{1, ..., n\}$, and let $\xi \in T_{x_\alpha}\mathbb{R}^m$ and consider
$$d_\alpha y_x(\xi),$$
where
$$y := y_\beta : (\mathbb{R}^m)^n \to \mathbb{R}^m.$$
To this end, if we let
$$\mu := \mathbb{E}[x], \qquad \sigma^2 := \mathbb{V}[x],$$
and consider $y$ written compositionally as
$$y : (\mathbb{R}^m)^n \times \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^m, \qquad y(x, \mu, \sigma^2) = (\sigma^2 + \vec{\epsilon})^{\odot -\frac{1}{2}} \odot (x_\beta - \mu),$$
then by the chain rule it follows that
$$d_\alpha y_x(\xi) = d_\alpha y_{(x,\mu,\sigma^2)}(\xi) + d_\mu y_{(x,\mu,\sigma^2)} \circ d_\alpha \mathbb{E}_x(\xi) + d_{\sigma^2} y_{(x,\mu,\sigma^2)} \circ d_\alpha \mathbb{V}_x(\xi).$$
Computing these differentials yields
$$\begin{aligned} d_\alpha y_{(x,\mu,\sigma^2)}(\xi) &= \delta_{\alpha\beta}(\sigma^2 + \vec{\epsilon})^{\odot -\frac{1}{2}} \odot \xi \\ d_\mu y_{(x,\mu,\sigma^2)}(\xi) &= -(\sigma^2 + \vec{\epsilon})^{\odot -\frac{1}{2}} \odot \xi \\ d_{\sigma^2} y_{(x,\mu,\sigma^2)}(\xi) &= -\frac{1}{2}(\sigma^2 + \vec{\epsilon})^{\odot -\frac{3}{2}} \odot (x_\beta - \mu) \odot \xi \\ d_\alpha \mathbb{E}_x(\xi) &= \frac{1}{n}\xi \\ d_\alpha \mathbb{V}_x(\xi) &= \frac{2}{n}(x_\alpha - \mu) \odot \xi. \end{aligned}$$
Substituting in these differentials, we see that
$$d_\alpha(y_\beta)_x(\xi) = \left[\delta_{\alpha\beta}(\sigma^2 + \vec{\epsilon})^{\odot -\frac{1}{2}} - \frac{1}{n}(\sigma^2 + \vec{\epsilon})^{\odot -\frac{1}{2}} - \frac{1}{n}(\sigma^2 + \vec{\epsilon})^{\odot -\frac{3}{2}} \odot (x_\beta - \mu) \odot (x_\alpha - \mu)\right] \odot \xi,$$

and noting that derivative only acts via the Hadamard-product, we may conclude that the reverse derivative coincides with the usual derivative, i.e.,

$$r_\alpha(y_\beta)_x \cong d_\alpha(y_\beta)_x,$$

after the usual identification of tangent spaces. To simplify this expression, we define the constant (with respect to the tangent space)

$$\theta = (\sigma^2 + \vec{\epsilon})^{\odot - \frac{1}{2}},$$

which leads us to write

$$d_\alpha(y_\beta)_x(\xi) = [\delta_{\alpha\beta}\theta - \frac{1}{n}\theta - \frac{1}{n}\theta \odot y_\alpha \odot y_\beta] \odot \xi.$$

Moreover, since

$$d(y_\beta)_x(\xi) = \sum_{\alpha=1}^{n} d_\alpha(y_\beta)_x(\xi_\alpha), \qquad \xi_\alpha \in T_{x_\alpha}\mathbb{R}^m,$$

it follows that for $\zeta_\beta \in T_{y_\beta}\mathbb{R}^m$, that

$$\begin{aligned}
\langle r(y_\beta)_x(\zeta_\beta), \xi\rangle_{(\mathbb{R}^m)^n} &= \langle \zeta_\beta, d(y_\beta)_x(\xi)\rangle_{T_{y_\beta}\mathbb{R}^m} \\
&= \left\langle \zeta_\beta, \sum_{\alpha=1}^{n} d_\alpha(y_\beta)_x(\xi_\alpha)\right\rangle_{T_{y_\beta}\mathbb{R}^m} \\
&= \sum_{\alpha=1}^{n} \langle r_\alpha(y_\beta)(\zeta_\beta), \xi_\alpha\rangle_{T_{x_\alpha}\mathbb{R}^m} \\
&= \left\langle \bigoplus_{\alpha=1}^{n} r_\alpha(y_\beta)_x(\zeta_\beta), \xi\right\rangle_{(\mathbb{R}^m)^n},
\end{aligned}$$

and hence that

$$r(y_\beta)_x(\zeta_\beta) = \bigoplus_{\alpha=1}^{n} r_\alpha(y_\beta)_x(\zeta_\beta).$$

Next, for $(x, \xi) \in T(\mathbb{R}^m)^n$ and $\zeta \in T_y(\mathbb{R}^m)^n$, we have that

$$\begin{aligned}
\langle rN_x(\zeta), \xi \rangle_{(\mathbb{R}^m)^n} &= \langle \zeta, dN_x(\xi) \rangle_{(\mathbb{R}^m)^n} \\
&= \left\langle \zeta, \bigoplus_{\beta=1}^{n} d(y_\beta)_x(\xi) \right\rangle_{(\mathbb{R}^m)^n} \\
&= \sum_{\beta=1}^{n} \langle \zeta_\beta, d(y_\beta)_x(\xi) \rangle_{T_{y_\beta}\mathbb{R}^m} \\
&= \sum_{\beta=1}^{n} \sum_{\alpha=1}^{n} \langle \zeta_\beta, d_\alpha(y_\beta)_x(\xi_\alpha) \rangle_{T_{y_\beta}\mathbb{R}^m} \\
&= \sum_{\beta=1}^{n} \sum_{\alpha=1}^{n} \langle r_\alpha(y_\beta)_x(\zeta_\beta), \xi_\alpha \rangle_{T_{x_\alpha}\mathbb{R}^m} \\
&= \sum_{\beta=1}^{n} \left\langle \bigoplus_{\alpha=1}^{n} r_\alpha(y_\beta)_x(\zeta_\beta), \xi \right\rangle_{(\mathbb{R}^m)^n} \\
&= \sum_{\beta=1}^{n} \langle r(y_\beta)_x(\zeta_\beta), \xi \rangle_{(\mathbb{R}^m)^n} \\
&= \left\langle \sum_{\beta=1}^{n} r(y_\beta)_x(\zeta_\beta), \xi \right\rangle_{(\mathbb{R}^m)^n} .
\end{aligned}$$

That is,

$$\begin{aligned}
rN_x(\zeta) &= \sum_{\beta=1}^{n} r(y_\beta)_x(\zeta_\beta) \\
&= \bigoplus_{\alpha=1}^{n} \left\{ \sum_{\beta=1}^{n} r_\alpha(y_\beta)_x(\zeta_\beta) \right\} \\
&= \bigoplus_{\alpha=1}^{n} \left\{ \sum_{\beta=1}^{n} \left[ \delta_{\alpha\beta}\theta \odot \zeta_\beta - \frac{1}{n}\theta \odot \zeta_\beta - \frac{1}{n}\theta \odot y_\alpha \odot y_\beta \odot \zeta_\beta \right] \right\} \\
&= \bigoplus_{\alpha=1}^{n} \left\{ \theta \odot \zeta_\alpha - \frac{1}{n}\theta \odot \sum_{\beta=1}^{n} \zeta_\beta - \frac{1}{n}\theta \odot y_\alpha \odot \sum_{\beta=1}^{n} y_\beta \odot \zeta_\beta \right\} \\
&= \bigoplus_{\alpha=1}^{n} \underbrace{\left\{ \theta \odot (\zeta e_\alpha) - \frac{1}{n}\theta \odot (\zeta\vec{1}) - \frac{1}{n}\theta \odot y_\alpha \odot (y \odot \zeta)\vec{1} \right\}}_{=r_\alpha N_x(\zeta)}.
\end{aligned}$$

We note here that $rN_x$ is a rank $(2,2)$-tensor, and as such we need to compute its components if we wish to implement this in python. To this end, let $\{E_i^j\}$ denote the basis for $\mathbb{R}^{m\times n}$, where

$$(E_i^j)_l^k = \delta_i^k \delta_l^j,$$

and let $\{\epsilon_j\}$, $\{e_j\}$ denote the standard bases for $\mathbb{R}^m$ and $\mathbb{R}^n$, respectively. We now compute

$$\begin{aligned}
rN_x(E_i^j) &= \bigoplus_{l=1}^{n} \left\{ \theta \odot (E_i^j e_l) - \frac{1}{n}\theta \odot (E_i^j\vec{1}) - \frac{1}{n}\theta \odot y_l \odot (y \odot E_i^j)\vec{1} \right\} \\
&= \bigoplus_{l=1}^{n} \left\{ \theta^k \delta_i^k \delta_l^j \epsilon_k - \frac{1}{n}\theta^k \delta_i^k \epsilon_k - \frac{1}{n}\theta^k y_l^k y_j^k \delta_i^k \epsilon_k \right\} \\
&= \bigoplus_{l=1}^{n} \theta^k \left\{ \delta_i^k \delta_l^j - \frac{1}{n}\delta_i^k(1 - y_l^k y_j^k) \right\} \epsilon_k \\
&= \theta^k \left[ \delta_i^k \delta_l^j - \frac{1}{n}\delta_i^k(1 - y_l^k y_j^k) \right] E_k^l \qquad \text{definition of direct sum} \\
&= \theta^k[\delta_i^k \delta_l^j - z^k{}_{lj}\delta_i^k]E_k^l,
\end{aligned}$$

that is, if $\zeta_j^i$ is a matrix, we yield the matrix

$$rN_x(\zeta) = \left[\theta^k[\delta_i^k \delta_l^j - z^k{}_{lj}\delta_i^k]\zeta_j^i\right]_l^k,$$

which is easily implemented in python via numpy's "einsum" function.

# References

[1] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.