Lecture notes

Introduction to Data Science and Deep Learning

Eran Treister
Computer Science Department,
Ben-Gurion University of the Negev.

Contents

1 Over-fitting, regularization and L-shaped curves

Overfitting occurs when we are trying to fit data models to sampled data, and have too many free parameters to do so. At some point, our model is in some sense too powerful, and is able to fit features in the data that are specific and are not relevant for the general population/data (inference). We will examine the phenomenon by using least squares to approximate an unknown function given noisy sample. Suppose we have the following function:

$$y_{true}(x) = \alpha_1 x + \alpha_2 \sin(10x) + \alpha_3 exp(2x). \tag{1}$$

Also, suppose that we are given with samples $\{(x_i, y_i)\}_{i=1}^n$ such that

$$y_i = y_{true}(x_i) + \varepsilon_i$$

where ε_i is some noise with mean 0. We will denote the given vector of y_i as $\mathbf{y}_{train}$. If we knew the ingredients of (1), we could just pick the mean squared error as our loss function, and estimate the coefficients α_i by least squares. That is:

$$\alpha_{oracle} = \arg\min_{\alpha} \|A_{true}\alpha - \mathbf{y}_{noisy}\|_2^2$$

where $A_{true} = [\ \mathbf{x} \mid \sin(10\mathbf{x}) \mid \exp(2\mathbf{x})\ ]$ is a matrix with the sampled functions in (1) as columns. $\mathbf{x}$ is the vector of the samples' locations. If the noise ε_i is rather uniform, this will lead to a very good recovery given that we have a reasonable number of points. Generally, the more points we have, the better will be the recovery as demonstrated below in terms of MSE. The MSE in the table is measured by a larger set of samples between the true and estimated coefficients.

Num samples	10	20	50	100	200	500	1000
MSE	7.02	3.5	2.4	2.5	2.3	1.6	0.72

However, as often happens, we do not know the true function to model our data, or such a function does not even exist. In that case we usually build our model using some predefined building blocks, which we connect through parameters. In the case of smooth function approximation, the polynomial basis if often a reasonable choice, although other bases like the Fourier basis is also an excellent choice. So we will first assume that our model

can approximate the true function

$$y_{true}(x) \approx y_k(x) = \sum_{i=0}^{k} \alpha_i x^i \tag{2}$$

and now we can also assume that

$$y_i = y_k(x) + \eta_i$$

and absorb the difference $y_{true}(x_i) - y_k(x_i)$ into η_i as model noise. So η_i is our way to include the model discrepancy and the sample noise together as just "noise". Given a polynomial degree k we can approximate the coefficients of y_k in (2), again via least squares:

$$\alpha_k = \arg\min_{\alpha} \|A_k \alpha_k - \mathbf{y}_{noisy}\|_2^2$$

where $A_k = [\ \mathbf{1} \mid \mathbf{x} \mid \mathbf{x}^2 \mid \cdots \mid \mathbf{x}^k\ ]$ is the matrix with the sampled polynomial basis functions as columns. The question is: how to choose the degree k? It turns out that this a tricky question, as any different true function in (1) yields a different answer. The optimal degree also depends on the sample noise ϵ_i and the number of sample points n.

To test that we will generate a new data $\mathbf{y}_{test}$ at the same way we generated $\mathbf{y}_{train}$ but with different noise, so that we are able distinguish what useful information can be obtained from the recovery, and what is just us fitting the noise. Figure 5 shows the actual approximation. The overfit is clear in the high degrees both in terms of the MSE and visually. In Fig. 2 we see two L-shaped curves, one that eventually decreases (the train) and one that increases (the test). In such scenarios, the turn of the L-shape in the training graph is usually the optimal point in the test, and is a good choice without actually having the test data. I does require doing quite a few experiments to get that graph.

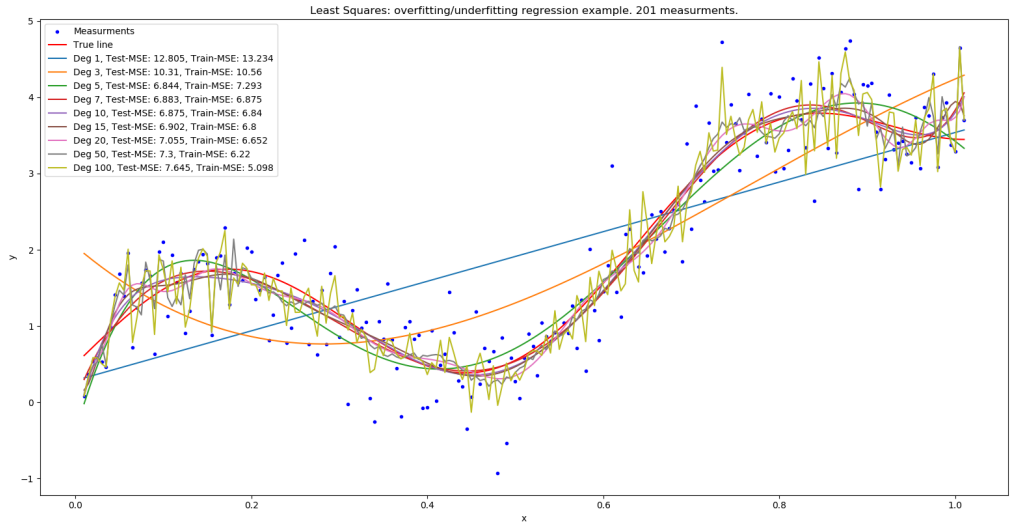

Figure 1: Approximation of the unknown function using polynomials: train and test MSE.

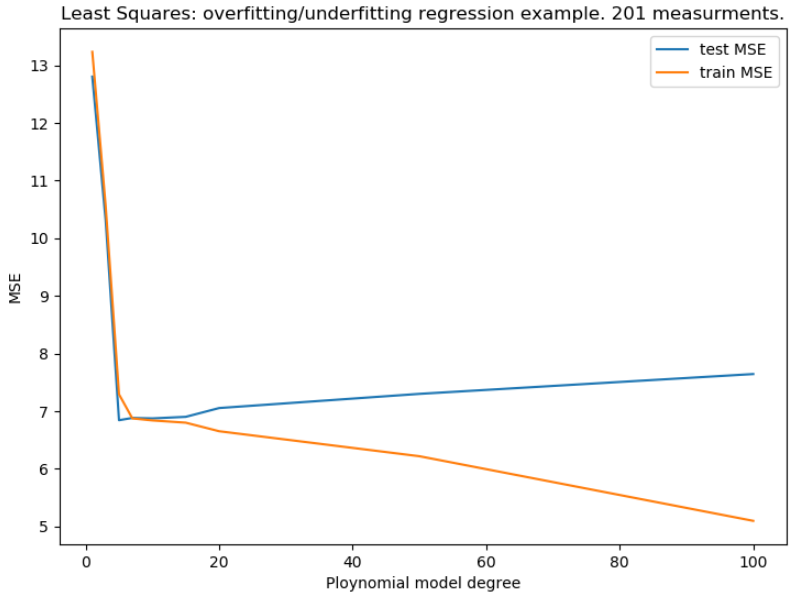

Figure 2: L-shaped curves: train vs test MSE.

```
## Generate a model of some function:
function generateTrueModel(x,alpha)
        A_true  = [x sin.(10.0.*x) exp.(2.0.*x)];
        ## Add noise so measurements will not exactly fit the model
        epsilon = 0.5*randn(length(x));
        y_true = A_true*alpha;
        y_noisy = y_true+epsilon;
        return A_true,y_true,y_noisy;
end

function estimateModel(y_noisy,A_basis)
        alpha_opt = (A_basis'*A_basis) \ A_basis'*y_noisy;  # equivalent: alpha_opt = A_basis\y_noisy;
        return alpha_opt;
end

x_full = collect(0.01:0.001:1.01);
A_true_full,y_true_full,y_noisy_full = generateTrueModel(x_full,[0.3;1.0;0.5])

x = collect(0.01:0.01:1.01);
A_true,y_true,y_noisy = generateTrueModel(x,[0.3;1.0;0.5])
noise_test = 0.5*randn(length(x));
alpha_star = estimateModel(y_noisy,A_true);

using PyPlot; close("all"); figure(1)
plot(x,y_noisy,".b");
plot(x,y_true,"-r");
plot(x,A_true*alpha_star,"-g");
title("Least Squares: overfitting/underfitting regression example");
legend(("Measurments","True line","Estimated line: oracle."));

for delta = [0.1;0.05;0.02;0.01;0.005;0.002;0.001]
        x = collect(0.01:delta:1.0);
        A_true,y_true,y_noisy = generateTrueModel(x,[0.3;1.0;0.5])
        alpha_star = estimateModel(y_noisy,A_true);
        println("num samples: ",length(x)," and MSE: ",norm(A_true_full*alpha_star-y_true_full));
end
```

```
include("OverfitExample.jl")
function generatePolyBasis(x::Array{Float64,1},degree)
        A = zeros(length(x),degree+1);
        A[:,1] .= 1.0;
        for k=1:degree
                A[:,k+1] .= A[:,k].*x;
        end
        # Pure numerical stuff see NLA: QR section:
        A = qr(A);
        A = Matrix(A.Q);
        return A;
end

polydegs = [1;3;5;7;10;15;20;50;100];
As                = Array{Array}(undef,length(polydegs));
As_full = Array{Array}(undef,length(polydegs));
alphas         = Array{Array}(undef,length(polydegs));
for k=1:length(polydegs)
        As[k]         = generatePolyBasis(x,polydegs[k]);
        alphas[k]     = estimateModel(y_noisy,As[k]);
end

deg_legend = Array{String}(undef,length(polydegs)+2);
deg_legend[1] = "Measurments"; deg_legend[2] = "True line";

figure(3); plot(x,y_noisy,".b"); plot(x,y_true,"-r"); xlabel("x"); ylabel("y")
title(string("Least Squares: overfitting/underfitting regression example. ",length(x)," measurments."));
test_mse = zeros(Float16,length(polydegs))
train_mse = zeros(Float16,length(polydegs))

for k=1:length(polydegs)
        yk = As[k]*alphas[k];
        plot(x,yk);
        test_mse[k] = norm(yk - y_true - noise_test);
        train_mse[k] = norm(yk - y_noisy);
        deg_legend[k+2] = string("Deg ",polydegs[k],", Test-MSE: ",test_mse[k],", Train-MSE: ",train_mse[k]);
        legend(tuple(deg_legend[1:k+2]...));
        pause(2.0);
        xlabel("x")
        ylabel("y")
end

figure(4)
plot(polydegs,test_mse); plot(polydegs,train_mse)
title(string("Least Squares: overfitting/underfitting regression example. ",length(x)," measurments."));
legend(("test MSE","train MSE"));
xlabel("Ploynomial model degree");ylabel("MSE")


# As_full[k]    = generatePolyBasis(x_full,polydegs[k]);
# figure(2)
# plot(x,y_noisy,".b");
# plot(x_full,y_true_full,"-r");
# plot(x_full,A_true_full*alpha_star,"-g");
# title("Least Squares: overfitting/underfitting regression example");
# legend(("Measurments","True line","Estimated line: oracle."));
```

Figure 3: MNIST data set: Given hand written images and their labels (a "training set" of 60,000 labeled images), learn how to classify letters.

2 Supervised Image Classification

One of the main and classical problems where Deep Learning is most successful is image classification, which has become one of the standard application to measure the quality of neural networks and related algorithms (optimization, architectures, etc.). We will use this problem as an example. In image classification we wish to predict a discrete value (sat, "0" or "1") for a given data, say an image. A famous data set is the MNIST data set illustrated in Fig. 3 where we need to learn a function (or, a computer program) who can say which number appears on a given image. We "teach" the program to predict the numbers for the images using examples of labeled images for which we train parameters. This is done by defining an objective function whose unknowns are the parameters that we wish to "learn" and minimizing this objective function using an iterative method.
A nice video to watch before digging into this, referring to the data in Figure 3:
`https://www.youtube.com/watch?v=aircAruvnKk`

2.1 Linear classifiers: Logistic regression

Logistic regression is a "linear classifier", which means that it predicts discrete values based on an inner product $\mathbf{x}^T\mathbf{w}$ (a linear function), more or less like least squares problems which

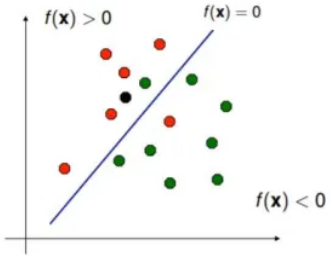

Figure 4: Two dimensional linear classifier. We have vectors $\mathbf{x} \in \mathbb{R}^2$, and we wish to distinguish between the red and green dots. We use the line defined by the function $f(\mathbf{x}) = w_1 x_1 + w_2 x_2 + b$, which is an equation of a straight line in space. If $f(\mathbf{x}) > 0$ we say that $\mathbf{x}$ is probably red, and if $f(\mathbf{x}) < 0$ we say that $\mathbf{x}$ is probably green. The model will be correct depending on the distribution of the dots in space, i.e., data dependent.

are also called "linear regression". The main difference is that based on this inner product (a continuous scalar), logistic regression predict a discrete value, 0 or 1. It does so by a function called "logistic" or "sigmoid" function

$$\sigma(\mathbf{x}^\top \mathbf{w}) = \frac{1}{1 + \exp(-\mathbf{x}^\top \mathbf{w})} \in [0, 1] \tag{3}$$

The function $\sigma(\mathbf{x}^\top \mathbf{w})$ may be interpreted as the probability of $\mathbf{x}$ to be 0 or 1, given the distribution parameter $\mathbf{w}$. If we decide that

$$\mathbb{P}(y = 1|\mathbf{x}) = \sigma(\mathbf{x}^\top \mathbf{w}), \quad \mathbb{P}(y = 0|\mathbf{x}) = 1 - \sigma(\mathbf{x}^\top \mathbf{w}) \tag{4}$$

then if $\sigma(\mathbf{x}^\top \mathbf{w})$ is close to 1, we will say that with high probability $y = 1$, and if not, then $y = 0$. The labels can be chosen the other way around, and then $\mathbf{w}$ that we will learn will just switch signs ($-\mathbf{w}$ will show probability 0 for label $y = 1$). We often refer to (3) as the "(probabilistic) model" in which we classify the images.

In "supervised learning" we are given with labeled data $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$, such that $\mathbf{x}_i \in \mathbb{R}^n$ and y_i are discrete categorial labels (may be even "cat" and "dog"). In logistic regression, we assume that we have only two labels (say, "cat" and "dog") and we wish to train $\mathbf{w}$ based

on minimizing the following "loss" function

$$F(\mathbf{w}) = -\frac{1}{m}\sum_{i=1}^{m}\left(I_{y_i=dog}\log(\mathbb{P}(\mathbf{x}_i = dog)) + I_{y_i=cat}\log(\mathbb{P}(\mathbf{x}_i = cat))\right) \quad (5)$$

$$= -\frac{1}{m}\sum_{i=1}^{m}\left(I_{y_i=dog}\log(\sigma(\mathbf{x}_i^\top\mathbf{w})) + I_{y_i=cat}\log(1-\sigma(\mathbf{x}_i^\top\mathbf{w}))\right) \quad (6)$$

which is called the "cross-entropy loss". $I_{y=dog}$ is an indicator function, that equals 1 if $y_i = dog$ and 0 otherwise. Once we minimize this loss function and get the estimated parameters $\mathbf{w}^*$, we can predict! Given a new image $\hat{\mathbf{x}}$, we compute $\sigma(\hat{\mathbf{x}}^\top\mathbf{w})$. If the value is larger than 0.5, then we'll predict that $\hat{\mathbf{x}}$ should be labeled as "dog", and otherwise, as a "cat".

The loss function in (5) can also be derived as the maximum likelyhood estimator (MLE) of $\mathbf{w}$ that corresponds to the Bernouli probability in (4), which can also be written as:

$$\mathbb{P}(y;q) = q^y \cdot (1-q)^{1-y}$$

where $q = \sigma(\mathbf{x}^\top\mathbf{w})$ takes the place of the Bernouli probability parameter, and $y = I_{y_i=dog}$ is compact notation for the indicator for the label. The probability parameter q is a function of a parameter $\mathbf{w}$ and data $\mathbf{x}$. Following the MLE recipe, we'll maximize over the likelihood of the data by taking the minus log of the likelihood and minimizing. This will yield:

$$\hat{\mathbf{w}}_{MLE} = \arg\max_{\mathbf{w}}\prod_{i=1}^{m}\mathbb{P}(y_i;q(\mathbf{w})) = \arg\min_{\mathbf{w}}\left\{-\sum_{i=1}^{m}\left(y_i\log(q(\mathbf{w})) + y_i\log(1-q(\mathbf{w}))\right)\right\}$$

this will also lead to a minimization corresponding to the probability $q(\mathbf{w}) = \sigma(\mathbf{x}^\top\mathbf{w})$, and corresponds to the objective in (5) if y_i are assumed to get values of 1 and 0 (the indicators).

We can write the objective in matrix-vector form. This way, it will be more efficient to compute the objective in languages that are vectorized like Matlab, Python, Julia or R. This is also taken into account in deep learning frameworks like PyTorch or TensorFlow etc.

Let $\mathbf{X} \in \mathbb{R}^{n\times m}$ be the data matrix

$$\mathbf{X} = [\mathbf{x}_1|\mathbf{x}_2|\mathbf{x}_3|...|\mathbf{x}_m],$$

and $\mathbf{c}_1, \mathbf{c}_2 \in \{0,1\}^m$ be the classes vectors as

$$\mathbf{c}_1 = [I_{y_1=dog}, I_{y_2=dog}, I_{y_3=dog}, ..., I_{y_m=dog}]^\top, \quad (7)$$

$$\mathbf{c}_2 = [I_{y_1=cat}, I_{y_2=cat}, I_{y_3=cat}, ..., I_{y_m=cat}]^\top. \quad (8)$$

The objective in (5) can be written in matrix form as

$$F(\mathbf{w}) = -\frac{1}{m}\left(\mathbf{c}_1^\top \log\left(\sigma(\mathbf{X}^\top\mathbf{w})\right) + \mathbf{c}_2^\top \log\left(1-\sigma(\mathbf{X}^\top\mathbf{w})\right)\right), \quad (9)$$

where the scalar function σ operates on each vector component separately. We may also write $\mathbf{c}_2 = 1 - \mathbf{c}_1$, but we keep it this way for the derivation of the softmax function that we will see next.

2.1.1 Adding the bias

In some sense, the logistic regression model (3) separates the domain into two sides by a linear function. If $\mathbf{x}^\top\mathbf{w} > 0$ the probability will come out larger than 0.5 and we will classify $\mathbf{x}$ as "1", while if $\mathbf{x}^\top\mathbf{w} < 0$, we will classify $\mathbf{x}$ as "0". This means that we are separating the entire $\mathbb{R}^n$ domain into two half-spaces by a linear function - all points $\mathbf{x}$ that lies above that plain will be classified together. As we know, a plain (linear function) is usually defined by $\mathbf{x}^\top\mathbf{w} + b$ with a free parameter $b \in \mathbb{R}$. Our case is no different, and in practice we need to add b to the parameters of the logistic regression classifier. This term is called "bias". To keep all the derivations above, one can add the bias artificially by adding a pixel of 1 to any image $\mathbf{x}$, and to add one parameter to $\mathbf{w}$.

2.2 Linear classifiers: Multinomial logistic (softmax) regression

The logistic regression objective was able to classify data images to two classes. What happens if we have more than two?? Say, "dog", "cat" and "horse" (in Fig 3 we have 10 labels)? The answer is the multinomial logistic regression which is also dubbed the "softmax" function, which is a simple generalization of the logistic regression model to l labels.

Assume again that we have the labeled data $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$, where this time $y_i \in \{1, ..., l\}$. We will again use the (3) model, but now we will have l weight vectors $\{\mathbf{w}_i\}_{i=1}^l$. For each image, we will have a probability of that image to be any of the labels (l probabilities). To

have them all sum to 1, we will define, somewhat similarly to (4)

$$\mathbb{P}(y=j|\mathbf{x}) = \frac{\exp(\mathbf{x}^\top \mathbf{w}_j)}{\sum_{i=1}^{l} \exp(\mathbf{x}^\top \mathbf{w}_i)} \in [0,1]. \tag{10}$$

The cross-entropy loss function can be written similarly to (5) and (9) (and is also a MLE corresponding to a "categorial distribution"):

$$\begin{aligned} F(\{\mathbf{w}_k\}_{k=1}^l) &= -\tfrac{1}{m}\sum_{i=1}^{m}\left(\sum_{k=1}^{l}(\mathbf{c}_k)_i \ \log\left(\left\{\sum_{j=1}^{l}\exp(\mathbf{x}_i^\top \mathbf{w}_j)\right\}^{-1}\exp(\mathbf{x}_i^\top \mathbf{w}_k)\right)\right) && (11)\\ &= -\tfrac{1}{m}\sum_{k=1}^{l}\mathbf{c}_k^\top \ \log\left(\operatorname{diag}\left\{\sum_{j=1}^{l}\exp(\mathbf{X}^\top \mathbf{w}_j)\right\}^{-1}\exp(\mathbf{X}^\top \mathbf{w}_k)\right). && (12)\end{aligned}$$

In softmax, one also need to add the bias terms, just like in logistic regression. We will have one bias term for each label.

Remark 1. *The diag$\{...\}^{-1}$ term above is written this way so that the expression fulfils the rule of linear algebra. Computing diag$(\mathbf{v})^{-1}\mathbf{u}$ "as is" in the code for two large vectors $\mathbf{v},\mathbf{u}$ is not efficient. To do so we may perform an element-wise division:*

$$\exp(\mathbf{X}^\top \mathbf{w}_k) \oslash \left\{\sum_{j=1}^{l}\exp(\mathbf{X}^\top \mathbf{w}_j)\right\}$$

. In code it is often looks like `u./v` *in most scientific languages (Matlab, Julia, etc.).*

Computing extreme exponentials "safely" The terms above should sum to 1. However, as $\mathbf{w}^\top\mathbf{x}$ is not bounded, the exponent may be computed by the computer as Inf (numerical overflow), leaving the whole computation undetermined. As it happens, we can instead compute

$$\frac{\exp(\mathbf{x}^\top \mathbf{w}_j)}{\sum_{i=1}^{l}\exp(\mathbf{x}^\top \mathbf{w}_i)} = \frac{\exp(\mathbf{x}^\top \mathbf{w}_j - \eta)}{\sum_{i=1}^{l}\exp(\mathbf{x}^\top \mathbf{w}_i - \eta)},$$

for some η, and in particular we can choose $\eta = \max_j\{\mathbf{x}^\top \mathbf{w}_j\}$, guaranteeing that the computation will be performed without overflow.

Remark 2. *The softmax objective* (11) *is redundant, as one computes a weight vector $\mathbf{w}_k$ for each label. Since we make sure all probabilities sum to 1, one can compute probabilities*

for all but 1 label, and that label completes the sum of probabilities to 1. That is the case in (5) *for two labels, where we learn only one weight vector* $\mathbf{w}$*. Note that comparing the probabilities* (3) *and* (10)*, once can see that* (3) *can be achieved using the formula in* (10) *with 2 vectors* $\mathbf{w}_j$ *where the second one is strictly zero. Hence, one can remove one weight vector from* (11)*, and set it to zero without loosing anything. Otherwise, we just have a redundant function with many equivalent local minima. That does not necessarily need to bother us, and in practice many people are using the redundant version of soft-max.*

2.3 Computing the derivatives of the linear classifiers

First, let us recall the following example:

Example 1. *(The Jacobian of* $\phi(\mathbf{x})$*, where* ϕ *is a scalar function) Suppose that* $\mathbf{f} = \phi(\mathbf{x})$*, where* $\phi : \mathbb{R} \to \mathbb{R}$ *is a scalar function, i.e.,* $f_i(\mathbf{x}) = \phi(x_i)$*. In this case the vector Taylor expansion is just a vector of one dimensional expansions.*

$$\delta\mathbf{f} = \phi(\mathbf{x} + \boldsymbol{\varepsilon}) - \phi_i(\mathbf{x}) \approx diag(\phi'(\mathbf{x}))\boldsymbol{\varepsilon} = diag(\phi'(\mathbf{x}))\delta\mathbf{x}.$$

This means that $\mathbf{J} = diag(\phi'(\mathbf{x}))$*, which is a diagonal matrix such that* $\mathbf{J}_{ii} = \phi'(x_i)$*.*

2.3.1 Computing the gradient of logistic regression

First,

$$\mathbf{c}^\top\phi(\mathbf{X}^\top(\mathbf{w} + \delta\mathbf{w})) \approx \mathbf{c}^\top\phi(\mathbf{X}^\top\mathbf{w}) + \mathbf{c}^\top\mathrm{diag}(\phi'(\mathbf{X}^\top\mathbf{w}))\mathbf{X}^\top\delta\mathbf{w}$$

and hence:

$$\nabla(\mathbf{c}^\top\phi(\mathbf{X}^\top\mathbf{w})) = \mathbf{X}\mathrm{diag}(\phi'(\mathbf{X}^\top\mathbf{w}))\mathbf{c} = \mathbf{X}(\phi'(\mathbf{X}^\top\mathbf{w}) \odot \mathbf{c}) \tag{13}$$

Next, it is not hard to show that for (3) we have

$$\sigma'(t) = \sigma(t)(1 - \sigma(t)), \quad (\log(\sigma(t)))' = 1 - \sigma(t), \quad (\log(1 - \sigma(t)))' = -\sigma(t)$$

In the first term of (9) we have $\phi_1(\mathbf{X}^\top\mathbf{w}) = \log(\sigma(\mathbf{X}^\top\mathbf{w}))$, and for the second one we have $\phi_2(\mathbf{X}\top\mathbf{w}) = \log(1 - \sigma(\mathbf{X}^\top\mathbf{w}))$. Hence we get

$$\begin{aligned}
\nabla_{\mathbf{w}} F &= -\frac{1}{m}\mathbf{X}\left[\phi_1'(\mathbf{X}^\top\mathbf{w}) \odot \mathbf{c}_1 + \phi_2'(\mathbf{X}^\top\mathbf{w}) \odot \mathbf{c}_2\right] && (14)\\
&= -\frac{1}{m}\mathbf{X}\left[(1-\sigma(\mathbf{X}^\top\mathbf{w})) \odot \mathbf{c}_1 - \sigma(\mathbf{X}^\top\mathbf{w}) \odot \mathbf{c}_2\right] && (15)\\
&= -\frac{1}{m}\mathbf{X}\left[(1-\sigma(\mathbf{X}^\top\mathbf{w})) \odot \mathbf{c}_1 - \sigma(\mathbf{X}^\top\mathbf{w}) \odot (1-\mathbf{c}_1)\right] && (16)\\
&= \frac{1}{m}\mathbf{X}\left[\sigma(\mathbf{X}^\top\mathbf{w}) - \mathbf{c}_1\right] && (17)
\end{aligned}$$

The Hessian The Hessian of (9), which is also the Jacobian of (14), can be computed using similar computations like (13). We get:

$$\nabla^2 F = \frac{1}{m}\mathbf{X}D\mathbf{X}^\top,$$

where $D = \sigma'(\mathbf{X}^\top\mathbf{w}) = \sigma(\mathbf{X}^\top\mathbf{w})(1-\sigma(\mathbf{X}^\top\mathbf{w}))$ is a diagonal matrix with only positive numbers on the diagonal. Overall, the Hessian must be a semi definite matrix and hence, the logistic regression objective is convex, and the strict convexity depends on whether the matrix $\mathbf{X}$ is full rank (rank at least n) or not. It can be shown using Taylor series that

$$F(\mathbf{w}+\delta\mathbf{w}) = \frac{1}{m}\|\mathbf{X}^\top\delta\mathbf{w} - \hat{\mathbf{g}}\|_D^2 + O(\|\delta\mathbf{w}\|^3),$$

for some vector $\hat{\mathbf{g}}$, and hence, SGD will asymptotically behave as it behaves for least squares problems, which we saw earlier.

2.3.2 The gradient of softmax regression w.r.t the weights

The computation of the gradient is quite similar to the computation of the logistic regression. We will use the following equality:

$$\frac{\partial}{\partial t_p}\sum_{k=1}^{l} a_k \log\left(\frac{\exp(t_k)}{\sum_j \exp(t_j)}\right) = a_p - \sum_{k=1}^{l} a_k \frac{\exp(t_p)}{\sum_j \exp(t_j)}$$

Using this equality and the fact that $\sum_k \mathbf{c}_k = 1$, we get

$$\begin{aligned}
\nabla_{\mathbf{w}_p} F &= -\frac{1}{m}\mathbf{X}\left[\mathbf{c}_p - \sum_{k=1}^{l} \operatorname{diag}\left\{\sum_j \exp(\mathbf{X}^\top \mathbf{w}_j)\right\}^{-1} \exp(\mathbf{X}^T \mathbf{w}_p)\mathbf{c}_k\right] \\
&= -\frac{1}{m}\mathbf{X}\left[\mathbf{c}_p - \operatorname{diag}\left\{\sum_j \exp(\mathbf{X}^\top \mathbf{w}_j)\right\}^{-1} \exp(\mathbf{X}^T \mathbf{w}_p)\sum_{k=1}^{l} \mathbf{c}_k\right] \\
&= \frac{1}{m}\mathbf{X}\left[\operatorname{diag}\left\{\sum_j \exp(\mathbf{X}^\top \mathbf{w}_j)\right\}^{-1} \exp(\mathbf{X}^\top \mathbf{w}_p) - \mathbf{c}_p\right].
\end{aligned}$$

2.3.3 The gradient of softmax regression w.r.t the data (needed for NN)

This derivative is necessary for combining the softmax loss function at the end of a forward pass of a neural network, described in the next section. Using similar computations as above we have in matrix form:

$$\nabla_{\mathbf{X}} F = \frac{1}{m}\mathbf{W}\left[\exp(\mathbf{W}^\top \mathbf{X}) \oslash \left(\sum_j \exp(\mathbf{w}_j^\top \mathbf{X})\right) - \mathbf{C}\right] \in \mathbb{R}^{n \times m}$$

where $\oslash$ is element-wise division (that also requires a broadcast or a `repmat`), $\mathbf{W} \in \mathbb{R}^{n \times l}$ is the matrix of all weights (arranged by columns), and $\mathbf{C} \in \mathbb{R}^{l \times m}$ is the matrix of all labels (arranged by rows):

$$\mathbf{C} = \begin{bmatrix} - & \mathbf{c}_1 & - \\ - & \mathbf{c}_2 & - \\ & \vdots & \\ - & \mathbf{c}_l & - \end{bmatrix}.$$

3 Neural Networks: definition.

A neural network is typically defined out of several building blocks that are chained one after the other, starting from a data measurement and ending with a loss function. A network typically have the following architecture:

$$F(\{\theta_l\}_{l=1}^{L}) = \frac{1}{m}\sum_{i=1}^{m}\ell(\theta^{(L)}, \mathbf{x}_i^{(L)}, y_i) \tag{18}$$

$$s.t \qquad \mathbf{x}_i^{(l+1)} = f_l(\theta^{(l)}, \mathbf{x}_i^{(l)}) \quad l = 1, ..., L-1, \quad i = 1, ..., m \tag{19}$$

where $\ell()$ is the softmax objective function defined in the previous section, and $\mathbf{x}_i^{(1)}$ denotes the given data sample i. We will denote by $\theta^{(l)}$ the set of weights of layer l that we need to learn. For example, for the last layer L we have the softmax weights:

$$\theta^{(L)} = \left\{\{\mathbf{w}_j^{(L)}\}_{j=1}^{n_{labels}} \in \mathbb{R}^n, \text{ and the bias vector } \mathbf{b}^{(L)} \in \mathbb{R}^{n_{labels}}.\right\}$$

For each data sample $\mathbf{x}_i$, the predicted label is the label with highest probability in the softmax function.

The step (also called "basic block" in common implementations)

$$\mathbf{x}_i^{(l+1)} = f_l(\theta^{(l)}, \mathbf{x}_i^{(l)}) \tag{20}$$

may be defined in various ways. The classical neural network is defined as

$$f(\theta, \mathbf{x}) = f(\{\mathbf{W}, \mathbf{b}\}, \mathbf{x}) = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}), \tag{21}$$

where σ is some non-linear "activation function". Such a step is demonstrated in the classical figure of a neural network in Fig. 10. Theoretically, any non-linear function can be used as activation. Typical activation functions may be the hyperbolic tangent $tanh$, the sigmoid that we've seen earlier, and the $ReLU(x) = \max(x, 0)$ function illustrated in Fig 9.

A more sophisticated step is the residual neural network (ResNet) defined as:

$$f(\{\mathbf{W}_1, \mathbf{W}_2, \mathbf{b}\}, \mathbf{x}) = \mathbf{x} + \mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}); \tag{22}$$

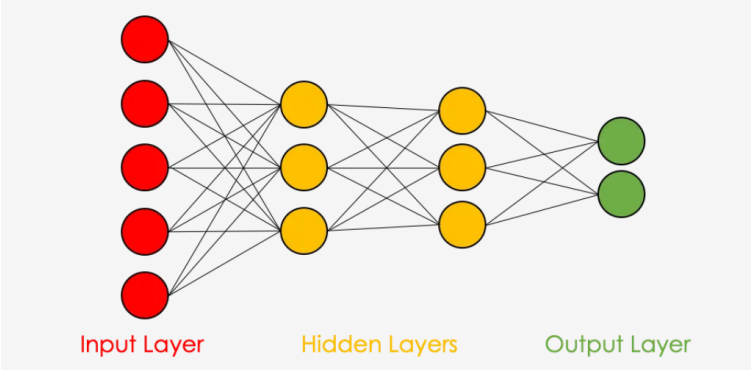

Figure 5: The schematic structure of a neural network with 4 layers. Each circle denotes a variable $x_j^{(l)}$ and the lines denote the connections between the layers imposed by multiplying each input $\mathbf{x}^{(l)}$ by the matrix $\mathbf{W}^{(l)}$ to get the output $\mathbf{x}^{(l+1)}$. The activation function is not illustrated in the figure.

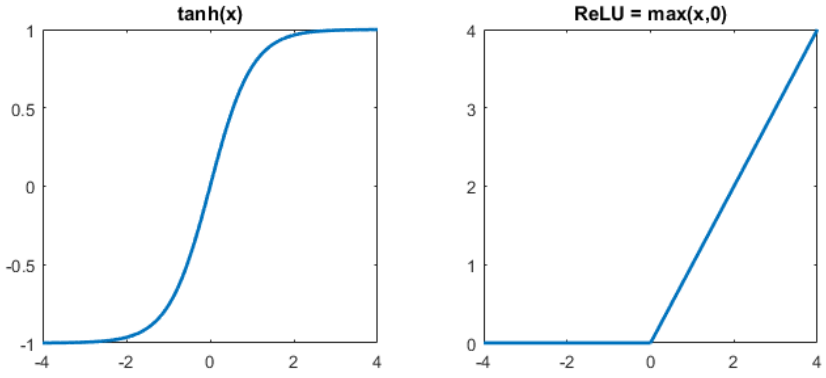

Figure 6: Commonly used activation functions $\sigma(x)$.

for which the dimension of $\mathbf{x}$ is the same as the dimension of f or

$$f(\{\mathbf{W}_1, \mathbf{W}_2, \mathbf{b}\}, \mathbf{x}) = \mathbf{W}_2\mathbf{x} + \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}); \tag{23}$$

for which the dimensions are different. There are several other type of steps in common NN's but we will stop here. In the language of Eq. (19), the unknowns that we have for the residual layers (or block of layers) $l = L-1, ..., 1$ are

$$\theta^{(l)} = \{\mathbf{W}_1^{(l)} \in \mathbb{R}^{n\times n}, \mathbf{W}_2^{(l)} \in \mathbb{R}^{n\times n}, \mathbf{b}^{(l)} \in \mathbb{R}^n\},$$

which needs to be learned as part of the training process (minimization of (19) for the training set).

Remark: the definition of layer Some people define each operation in the function above as "layer" (for example, the addition of bias, the multiplication of $\mathbf{W}$, the non-linear function σ etc, each one is a different "layer"). The whole residual network step is then defined as a block of layers.

3.1 NNs as function interpolation: The Universal approximation theorem

Deep learning can be viewed as a way to approximate a function: $\Psi(\mathbf{x}) = y$ based on the labeled data $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$. This task of defining a function using building blocks and parameters to approximate another unknown function which is given by samples is called interpolation. That is, we define $\Psi(\mathbf{x}) \approx \hat{\Psi}(\mathbf{x}, \theta)$ by satisfying:

$$\hat{\Psi}(\mathbf{x}_i, \theta) \approx y_i, \quad i = 1, ..., m.$$

The most classical type of interpolation is the polynomial interpolation for which the parameters are the coefficients of polynomials, or the Fourier series where the parameters are the coefficients of trigonometric functions. In NNs, the approximating function is defined as the output of the network, and this approximation is possible and accurate thanks to the following theorem:

A multi-layer perceptron (MLP - a matrix multiplication + non-linear activation) with

a single hidden layer containing a finite number of neurons can approximate any continuous function f on compact subsets of $\mathbb{R}^n$ (in finite dimension, a set is said to be compact if it is closed and bounded.). Therefore, the class of deep neural networks is a universal approximator iff the activation function σ is non-linear.

The main take-away of this theorem is that deep learning allows us to solve any problem whose solution can be mathematically expressed as a function, if we have sufficient amount of data and can find good parameters for the associated interpolation problem. However, the theorem does not state how many variables should be in that hidden layer, and in practice, many variables are necessary making the approximation computationally expensive. This is often compensated by using more and more layers (a.k.a deep networks).

4 Training of a Neural Network: minimizing the loss via back-propagation.

The constrained objective (19) can be minimized in various ways. The most common way to solve it is by SGD. To apply SGD, we need to be able to compute the gradient of the network w.r.t all the parameters θ. From practical reasons, the constraint is handled by elimination, removing $\mathbf{x}^{(l)}$ as unknowns, as it is just defined by the application (the forward pass) of the network. We get a nested chain of functions of the type

$$F(\{\theta^{(i)}\}_{i=1}^{3}) = f_3(x^{(3)}, \theta^{(3)}) = f_3(f_2(x^{(2)}, \theta^{(2)}), \theta^{(3)}) = f_3(f_2(f_1(x^{(1)}, \theta^{(1)}), \theta^{(2)}), \theta^{(3)}), \quad (24)$$

where in this example, $L = 3$, and $x^{(1)}$ is the given data.

Let's assume that all the quantities above are scalars, i.e.,

$$f_i(x, \theta) : (\mathbb{R}, \mathbb{R}) \to \mathbb{R}$$

so each derivative will also be a scalar. We wish to get the derivative of f_3 in (25) with respect to $\theta^{(1)}, \theta^{(2)}, \theta^{(3)}$. We get a special structure using inner derivatives:

$$\frac{\partial F}{\partial \theta^{(3)}} = \frac{\partial f_3}{\partial \theta} \quad (25)$$

$$\frac{\partial F}{\partial \theta^{(2)}} = \frac{\partial f_3}{\partial x} \cdot \frac{\partial f_2}{\partial \theta} \quad (26)$$

$$\frac{\partial F}{\partial \theta^{(1)}} = \frac{\partial f_3}{\partial x} \cdot \frac{\partial f_2}{\partial x} \cdot \frac{\partial f_1}{\partial \theta}. \quad (27)$$

The order of multiplications in (26) could have been different as these are all scalars, but it turns out (as we will see later) that the right way to put it is

$$\frac{\partial F}{\partial \theta^{(3)}} = \frac{\partial f_3}{\partial \theta} \quad (28)$$

$$\frac{\partial F}{\partial \theta^{(2)}} = \frac{\partial f_2}{\partial \theta} \cdot \frac{\partial f_3}{\partial x} \quad (29)$$

$$\frac{\partial F}{\partial \theta^{(1)}} = \frac{\partial f_1}{\partial \theta} \cdot \frac{\partial f_2}{\partial x} \cdot \frac{\partial f_3}{\partial x}. \quad (30)$$

This is the right order if all functions where vector functions of vector unknowns. To compute

(25) we start from f_1, and end at f_3. That is a **forward pass**. To compute the gradient of F, we start from f_3 and propagate backwards all the way to f_1. That is called the **backpropagation pass**.

To illustrate the right order we will rewrite (25) in vector form for $L = 2$, but also include the scalar loss function at the end. We denote: $\ell = \ell(\mathbf{x}, \boldsymbol{\theta})$, and $\mathbf{f}_i = \mathbf{f}_i(\mathbf{x}, \boldsymbol{\theta})$. That is, each function $\mathbf{f}_i$ is defined for two unknowns and is not "aware" of its location in the network.

$$F(\{\boldsymbol{\theta}^{(i)}\}_{i=1}^{3}) = \ell(\mathbf{f}_2(\mathbf{x}^{(2)}, \boldsymbol{\theta}^{(2)}), \boldsymbol{\theta}^{(3)}) = \ell(\mathbf{f}_2(\mathbf{f}_1(\mathbf{x}^{(1)}, \boldsymbol{\theta}^{(1)}), \boldsymbol{\theta}^{(2)}), \boldsymbol{\theta}^{(3)}). \tag{31}$$

We wish to compute the gradient of F with respect to all the parameters (in order to use an optimization algorithm):

$$\nabla F = \begin{bmatrix} \nabla_{\boldsymbol{\theta}^{(3)}} F \\ \nabla_{\boldsymbol{\theta}^{(2)}} F \\ \nabla_{\boldsymbol{\theta}^{(1)}} F \end{bmatrix}$$

Since $\boldsymbol{\theta}^{(3)}$ only involves with ℓ, then the gradient with respect to $\boldsymbol{\theta}^{(3)}$ is simply:

$$\nabla_{\boldsymbol{\theta}^{(3)}} F = \nabla_{\boldsymbol{\theta}} \ell.$$

The rest of the gradient requires the chain rule, because it involves with $\mathbf{f}_i$ inside ℓ. We know that simple Taylor series is given by

$$\ell(\mathbf{x} + \delta\mathbf{x}, \boldsymbol{\theta}) \approx \ell(\mathbf{x}, \boldsymbol{\theta}) + \delta\mathbf{x}^T \nabla_{\mathbf{x}} \ell. \tag{32}$$

Also, for every $\mathbf{x}$ and $\boldsymbol{\theta}$

$$\mathbf{f}_2(\mathbf{x}, \boldsymbol{\theta} + \delta\boldsymbol{\theta}) \approx \mathbf{f}_2(\mathbf{x}, \boldsymbol{\theta}) + \frac{\partial \mathbf{f}_2}{\partial \boldsymbol{\theta}} \delta\boldsymbol{\theta}.$$

and

$$\mathbf{f}_2(\mathbf{x} + \delta\mathbf{x}, \boldsymbol{\theta}) \approx \mathbf{f}_2(\mathbf{x}, \boldsymbol{\theta}) + \frac{\partial \mathbf{f}_2}{\partial \mathbf{x}} \delta\mathbf{x}. \tag{33}$$

Now, assume that instead of a general input $\mathbf{x} + \delta\mathbf{x}$, we put the input from the previous layer in (34):

$$\mathbf{f}_1(\mathbf{x}^{(1)}, \boldsymbol{\theta}^{(1)} + \delta\boldsymbol{\theta}^{(1)}) = \mathbf{f}_1(\mathbf{x}^{(1)}, \boldsymbol{\theta}^{(1)}) + \delta\mathbf{f}_1 \approx \mathbf{f}_1(\mathbf{x}^{(1)}, \boldsymbol{\theta}^{(1)}) + \frac{\partial \mathbf{f}_1}{\partial \boldsymbol{\theta}} \delta\boldsymbol{\theta}^{(1)}.$$

After placing this in (34) instead of $x + \delta\mathbf{x}$ we get:

$$\begin{aligned}\mathbf{f}_2(\mathbf{f}_1(\mathbf{x}^{(1)}, \boldsymbol{\theta}^{(1)} + \delta\boldsymbol{\theta}^{(1)}), \boldsymbol{\theta}^{(2)}) &\approx \mathbf{f}_2(\mathbf{f}_1(\mathbf{x}^{(1)}, \boldsymbol{\theta}^{(1)}), \boldsymbol{\theta}^{(2)}) + \frac{\partial \mathbf{f}_2}{\partial \mathbf{x}}\delta\mathbf{f}_1 \\ &\approx \mathbf{f}_2(\mathbf{f}_1(\mathbf{x}^{(1)}, \boldsymbol{\theta}^{(1)}), \boldsymbol{\theta}^{(2)}) + \frac{\partial \mathbf{f}_2}{\partial \mathbf{x}}\frac{\partial \mathbf{f}_1}{\partial \boldsymbol{\theta}}\delta\boldsymbol{\theta}^{(1)}.\end{aligned}$$

So, overall $\delta\mathbf{f}_2 = \frac{\partial \mathbf{f}_2}{\partial \mathbf{x}}\frac{\partial \mathbf{f}_1}{\partial \boldsymbol{\theta}}\delta\boldsymbol{\theta}^{(1)}$. Now we plug in (33):

$$\begin{aligned}\ell(\mathbf{f}_2 + \delta\mathbf{f}_2, \boldsymbol{\theta}^{(3)}) &\approx \ell(\mathbf{f}_2, \boldsymbol{\theta}^{(3)}) + \delta\mathbf{f}_2^T\nabla_{\mathbf{x}}\ell \\ &= \ell(\mathbf{f}_2, \boldsymbol{\theta}^{(3)}) + \left(\frac{\partial \mathbf{f}_2}{\partial \mathbf{x}}\frac{\partial \mathbf{f}_1}{\partial \boldsymbol{\theta}}\delta\boldsymbol{\theta}^{(1)}\right)^T\nabla_{\mathbf{x}}\ell \\ &= \ell(\mathbf{f}_2, \boldsymbol{\theta}^{(3)}) + \left(\delta\boldsymbol{\theta}^{(1)}\right)^\top \frac{\partial \mathbf{f}_1}{\partial \boldsymbol{\theta}}^T\frac{\partial \mathbf{f}_2}{\partial \mathbf{x}}^T\nabla_{\mathbf{x}}\ell\end{aligned}$$

From here, the gradient of (32) is

$$\begin{aligned}\nabla_{\boldsymbol{\theta}^{(3)}}F &= \nabla_{\boldsymbol{\theta}}\ell \\ \nabla_{\boldsymbol{\theta}^{(2)}}F &= \frac{\partial \mathbf{f}_2}{\partial \boldsymbol{\theta}}^T\nabla_{\mathbf{x}}\ell \\ \nabla_{\boldsymbol{\theta}^{(1)}}F &= \frac{\partial \mathbf{f}_1}{\partial \boldsymbol{\theta}}^T\frac{\partial \mathbf{f}_2}{\partial \mathbf{x}}^T\nabla_{\mathbf{x}}\ell\end{aligned}$$

If we would have had another layer, say, layer 0, we would have:

$$\nabla_{\boldsymbol{\theta}^{(0)}}F = \frac{\partial \mathbf{f}_0}{\partial \boldsymbol{\theta}}^T\frac{\partial \mathbf{f}_1}{\partial \mathbf{x}}^T\frac{\partial \mathbf{f}_2}{\partial \mathbf{x}}^T\nabla_{\mathbf{x}}\ell$$

To sum up, for each layer $f(\mathbf{x}, \boldsymbol{\theta})$, we need to compute the Jacobian (with respect to each of the parameters) transpose multiplied with some vector. We will define this operation for our residual network.

4.1 Computing derivatives of Neural Networks

As we saw from the derivations above, we need to be able to handle the Jacobians of all the ingredients in the network. But, luckily, we do not need to explicitly form Jacobian matrices (that will be too big and expensive). By (35), all we need is to be able to compute

the multiplication of the transposed Jacobian with some vector coming from the front of the network. We'll generally denote this vector as $\mathbf{v}$.

4.1.1 The derivatives of matrix-vector and matrix-matrix multiplications

We will start with a rather simple auxiliary computation that will help us later. Assume

$$\mathbf{y} = \mathbf{W}\mathbf{x}; \quad \mathbf{W} \in \mathbb{R}^{t\times n}; \quad \mathbf{y} \in \mathbb{R}^t; \quad \mathbf{x} \in \mathbb{R}^n$$

We've already seen earlier in the course that $\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{W}$, and hence simply $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}^T = \mathbf{W}^T$.

Now we'll examine the case of the derivative with respect to $\mathbf{W}$:

$$\mathbf{y} + \delta\mathbf{y} = (\mathbf{W} + \delta\mathbf{W})\mathbf{x} \Rightarrow \delta\mathbf{y} = \delta\mathbf{W}\mathbf{x}.$$

Using different writing, we wish to have

$$\delta\mathbf{W}\mathbf{x} \equiv \frac{\partial \mathbf{y}}{\partial \mathbf{W}}\text{vec}(\delta\mathbf{W}),$$

where $\frac{\partial \mathbf{y}}{\partial \mathbf{W}} \in \mathbb{R}^{t\times tn}$ is a Jacobian matrix and the operation vec($\mathbf{W}$) is the concatanation of the columns of $\mathbf{W}$ one after the other into a long vector of size tn (also known as column-stack representation, achieved by the operator [:] in Julia). We need to know how to define $\frac{\partial \mathbf{y}}{\partial \mathbf{W}}$ and to compute its transpose times a vector.

To obtain this, we will start with a few definitions to let us work with matrices and their derivatives. To deal with matrices as unknowns, we will transfer them into vectors first, compute the derivative and transfer them back. For that we have the following definition:

Kronacker product: If $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{p\times q}$ the Kronacker product $A \otimes B$ is the $mp \times nq$ block matrix

$$\mathbf{A}\otimes\mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix},$$

In particular we will use the following identities:

$$(A \otimes B)^T = A^T \otimes B^T \tag{34}$$

$$AXB = C \Rightarrow (B^T \otimes A)\text{vec}(X) = \text{vec}(C) \tag{35}$$

Back to our auxiliary computation:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{W}}\text{vec}(\delta \mathbf{W}) = \delta \mathbf{W}\mathbf{x},$$

then by (36) we have

$$I\delta \mathbf{W}\mathbf{x} = (\mathbf{x}^T \otimes I)\text{vec}(\delta \mathbf{W}) \Rightarrow \frac{\partial \mathbf{y}}{\partial \mathbf{W}} = \mathbf{x}^T \otimes I. \tag{36}$$

By the identity in (35) we have that the Jacobian trasposed times some vector is given by:

$$(\mathbf{x}^T \otimes I)^T \mathbf{v} = (\mathbf{x} \otimes I)\mathbf{v} = \mathbf{v}\mathbf{x}^T \in \mathbb{R}^{t \times n} \tag{37}$$

where $\mathbf{v}$ is some vector of the same size of $\mathbf{y} = \mathbf{W}\mathbf{x}$. $\mathbf{v}\mathbf{x}^T$ is the same size of $\mathbf{W}$ without column stack. Like in the case of Logistic Regression, if we have many data samples $\mathbf{x}_i$ stacked in a matrix column-wise:

$$\mathbf{X} = [\mathbf{x}_1|\mathbf{x}_2|\mathbf{x}_3|...|\mathbf{x}_m],$$

and we wish to take the derivative of $\mathbf{Y} = \mathbf{W}\mathbf{X}$ with respect to $\mathbf{W}$. Similarly to before, the Jacobian transposed times a matrix $\mathbf{V}$ of the same size as $\mathbf{Y}$ is given by:

$$(\mathbf{X} \otimes I)\mathbf{V} =_{\text{after de-column-stacking}} \sum_i \mathbf{v}_i \mathbf{x}_i^\top = \mathbf{V}\mathbf{X}^\top.$$

4.1.2 The derivatives of standard neural networks

Let's look at our network step:

$$\mathbf{y} = f(\theta, \mathbf{x}) = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}).$$

In our backpropagation, we should multiply different vectors $\mathbf{v}$ of size similar to $\mathbf{y}$ with the transpose of the Jacobian with respect to each of the weights and unknowns $\mathbf{x} \in \mathbb{R}^n, \mathbf{W} \in \mathbb{R}^{k\times n}, \mathbf{b} \in \mathbb{R}^k$.

The derivative of $\mathbf{y} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$ with respect to $\mathbf{b} \in \mathbb{R}^k$:

$$\begin{aligned}
\mathbf{y} + \delta\mathbf{y} &= \sigma(\mathbf{W}\mathbf{x} + \mathbf{b} + \delta\mathbf{b}) \\
&\approx \mathbf{y} + \mathrm{diag}(\sigma'(\mathbf{W}\mathbf{x} + \mathbf{b}))\delta\mathbf{b} \\
&\Rightarrow \frac{\partial \mathbf{y}}{\partial \mathbf{b}} = \mathrm{diag}(\sigma'(\mathbf{W}\mathbf{x} + \mathbf{b})) \in \mathbb{R}^{k\times k} \\
&\Rightarrow \frac{\partial \mathbf{y}}{\partial \mathbf{b}}^T \mathbf{v} = \mathrm{diag}(\sigma'(\mathbf{W}\mathbf{x} + \mathbf{b}))\mathbf{v} = \sigma'(\mathbf{W}\mathbf{x} + \mathbf{b}) \odot \mathbf{v}.
\end{aligned}$$

The derivative of $\mathbf{y} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$ with respect to $\mathbf{W} \in \mathbb{R}^{k\times n}$:

$$\begin{aligned}
\mathbf{y} + \delta\mathbf{y} &= \sigma((\mathbf{W} + \delta\mathbf{W})\mathbf{x} + \mathbf{b}) \\
&\approx \mathbf{y} + \mathrm{diag}(\sigma'(\mathbf{W}\mathbf{x} + \mathbf{b}))\delta\mathbf{W}\mathbf{x} \\
&\Rightarrow \frac{\partial \mathbf{y}}{\partial \mathbf{W}} = \mathrm{diag}(\sigma'(\mathbf{W}\mathbf{x} + \mathbf{b}))(\mathbf{x}^T \otimes I) \in \mathbb{R}^{k\times kn} \\
&\Rightarrow \frac{\partial \mathbf{y}}{\partial \mathbf{W}}^T \mathbf{v} = (\mathbf{x}^T \otimes I)^T \mathrm{diag}(\sigma'(\mathbf{W}\mathbf{x} + \mathbf{b}))\mathbf{v} = (\sigma'(\mathbf{W}\mathbf{x} + \mathbf{b}) \odot \mathbf{v})\mathbf{x}^T.
\end{aligned}$$

Here we used (37), and to multiply the Jacobian transposed with a vector $\mathbf{v}$, we followed (38).

The derivative $\mathbf{y} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$ with respect to $\mathbf{x}$:

$$\begin{aligned}
\mathbf{y} + \delta\mathbf{y} &= \sigma(\mathbf{W}(\mathbf{x} + \delta\mathbf{x}) + \mathbf{b}) \\
&\approx \mathbf{y} + \mathrm{diag}(\sigma'(\mathbf{W}\mathbf{x} + \mathbf{b}))\mathbf{W}\delta\mathbf{x} \\
&\Rightarrow \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathrm{diag}(\sigma'(\mathbf{W}\mathbf{x} + \mathbf{b}))\mathbf{W} \in \mathbb{R}^{k\times n} \\
&\Rightarrow \frac{\partial \mathbf{y}}{\partial \mathbf{x}}^T \mathbf{v} = \mathbf{W}^T(\sigma'(\mathbf{W}\mathbf{x} + \mathbf{b}) \odot \mathbf{v})
\end{aligned}$$

4.1.3 The derivatives of residual networks

Let's look again at our residual network step:

$$\mathbf{y} = f(\theta, \mathbf{x}) = \mathbf{x} + \mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}).$$

In our backpropagation, we should multiply different vectors $\mathbf{v}$ of size similar to $f(\theta, \mathbf{x})$ with the transpose of the Jacobian with respect to each of the weights and unknowns $\mathbf{x}, \mathbf{W}_2, \mathbf{W}_1, \mathbf{b}$.

The derivative with respect to b:

$$\begin{aligned} \mathbf{y} + \delta\mathbf{y} &= \mathbf{x} + \mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b} + \delta\mathbf{b}) \\ &\approx \mathbf{y} + \mathbf{W}_2\mathrm{diag}(\sigma'(\mathbf{W}_1\mathbf{x} + \mathbf{b}))\delta\mathbf{b} \\ &\Rightarrow \frac{\partial\mathbf{y}}{\partial\mathbf{b}} = \mathbf{W}_2\mathrm{diag}(\sigma'(\mathbf{W}_1\mathbf{x} + \mathbf{b})) \in \mathbb{R}^{n\times n}. \end{aligned}$$

The derivative with respect to $\mathbf{W}_1$:

$$\begin{aligned} \mathbf{y} + \delta\mathbf{y} &= \mathbf{x} + \mathbf{W}_2\sigma((\mathbf{W}_1 + \delta\mathbf{W}_1)\mathbf{x} + \mathbf{b}) \\ &\approx \mathbf{y} + \mathbf{W}_2\mathrm{diag}(\sigma'(\mathbf{W}_1\mathbf{x} + \mathbf{b}))\delta\mathbf{W}_1\mathbf{x} \\ &\Rightarrow \frac{\partial\mathbf{y}}{\partial\mathbf{W}_1} = \mathbf{W}_2\mathrm{diag}(\sigma'(\mathbf{W}_1\mathbf{x} + \mathbf{b}))(\mathbf{x}^T \otimes I) \in \mathbb{R}^{n\times n^2}. \end{aligned}$$

In the last equation we used (37), and to multiply the transpose of this matrix with a vector $\mathbf{v}$, we need to follow (38).

The derivative with respect to $\mathbf{W}_2$:

$$\begin{aligned} \mathbf{y} + \delta\mathbf{y} &= \mathbf{x} + (\mathbf{W}_2 + \delta\mathbf{W}_2)\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}) \\ &= \mathbf{y} + \delta\mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}) \\ &\Rightarrow \frac{\partial\mathbf{y}}{\partial\mathbf{W}_2} = (\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b})^T \otimes I) \in \mathbb{R}^{n\times n^2}. \end{aligned}$$

Here, again, we used (37), only with respect to $\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b})$ instead of $\mathbf{x}$. To multiply the transpose of this matrix with a vector $\mathbf{v}$, we need to follow (38).

The derivative with respect to x:

$$\begin{aligned} \mathbf{y}+\delta\mathbf{y} &= \mathbf{x}+\delta\mathbf{x}+\mathbf{W}_2\sigma(\mathbf{W}_1(\mathbf{x}+\delta\mathbf{x})+\mathbf{b}) \\ &\approx \mathbf{y}+\delta\mathbf{x}+\mathbf{W}_2\mathrm{diag}(\sigma'(\mathbf{W}_1\mathbf{x}+\mathbf{b}))\mathbf{W}_1\delta\mathbf{x} \\ &\Rightarrow \frac{\partial\mathbf{y}}{\partial\mathbf{x}} = I+\mathbf{W}_2\mathrm{diag}(\sigma'(\mathbf{W}_1\mathbf{x}+\mathbf{b}))\mathbf{W}_1 \in \mathbb{R}^{n\times n}. \end{aligned}$$

4.2 Custom implementation of backward in Pytorch

Class discussion:

MyRelu:

`https://pytorch.org/tutorials/beginner/examples_autograd/two_layer_net_custom_function.html`

Linear:

`https://pytorch.org/docs/stable/notes/extending.html`

4.3 Gradient and Jacobian verification

Each time we compute a non-linear function and its derivative, we need to make sure that our derivations are correct. It is better not to test this though the whole optimization process of our system, and do it part by part. This way, (1) we will be able to identify the errors in the right places, and (2) we won't be confused in the case where our optimization algorithm really does not supposed to work (the lack of minimization is a "feature instead of a bug").

The gradient test: The simplest case is achieved for the gradient of a scalar function. Let $\mathbf{d}$ be a random vector, such that $\|\mathbf{d}\| = O(1)$. Then, we know that

$$f(\mathbf{x}+\epsilon\mathbf{d}) = f(\mathbf{x}) + \epsilon\mathbf{d}^T\nabla f + O(\epsilon^2)$$

Suppose that we have a code $\mathtt{grad}(\mathbf{x})$ for computing $\nabla f(\mathbf{x})$. Then, we can compare

$$|f(\mathbf{x}+\epsilon\mathbf{d}) - f(\mathbf{x})| = O(\epsilon) \tag{38}$$

versus

$$|f(\mathbf{x}+\epsilon\mathbf{d}) - f(\mathbf{x}) - \epsilon\mathbf{d}^T\mathtt{grad}(\mathbf{x})| = O(\epsilon^2). \tag{39}$$

We will test these two values for decreasing values of ϵ. For example $\epsilon_i = (0.5)^i \epsilon_0$ for some ϵ_0. The values of (39) should decrease linearly (be smaller by a factor of two for each iterate of i), and the values of (40) should (if the code is right) decrease quadratically (be smaller by a factor of four for each iterate of i).

The Jacobian test: The same technique can be used to test the computation of the Jacobian. Here, we do not necessarily compute the matrix explicitly, but compute its multiplication with a vector. Assume that you have the code $\texttt{JacMV}(\mathbf{x}, \mathbf{v})$, that computes the multiplication of the Jacobian with a vector $\mathbf{v}$, i.e., $\frac{\partial f}{\partial \mathbf{x}}\mathbf{v}$. The derivative is computed at the point $\mathbf{x}$. Similarly to before, we will use a vector $\mathbf{v} = \epsilon \mathbf{d}$

$$\|f(\mathbf{x} + \epsilon \mathbf{d}) - f(\mathbf{x})\| = O(\epsilon)$$

versus

$$\|f(\mathbf{x} + \epsilon \mathbf{d}) - f(\mathbf{x}) - \texttt{JacMV}(\mathbf{x}, \epsilon \mathbf{d})\| = O(\epsilon^2).$$

The test is exactly the same procedure as the gradient test.

The transpose test: In this test we assume that we have a routine $\texttt{JacTMV}(\mathbf{x}, \mathbf{v})$ that computes the multiplication $\left(\frac{\partial f}{\partial \mathbf{x}}\right)^T \mathbf{v}$. We will assume that our `JacMV` function is correct, and passed the Jacobian test. Then we will use the equality

$$\mathbf{u}^T \mathbf{J} \mathbf{v} = \mathbf{v}^T \mathbf{J}^T \mathbf{u},$$

which holds for any two random vectors $\mathbf{u}, \mathbf{v}$. To pass the test, we will verify that

$$|\mathbf{u}^T \texttt{JacMV}(\mathbf{x}, \mathbf{v}) - \mathbf{v}^T \texttt{JacTMV}(\mathbf{x}, \mathbf{u})| < \epsilon,$$

where ϵ is the machine precision.

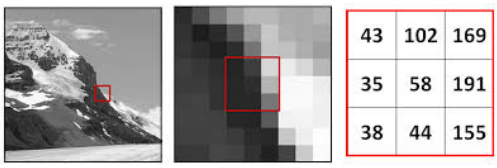

Figure 7: Digital image representation.

5 Convolutional Neural Networks (CNNs)

In recent years, Convolutional Neural Networks (CNNs) have become a force for tackling computer vision applications. CNNs are quite similar to standard NN's in some way, only their weight matrices have special structure - one of a spatial convolution. As we will see later, a spatial convolution is a sparse matrix with a specific structure, but it is a matrix. Therefore, a CNN is a special form of a standard NN. The problem of applying a CNN on images is that the images may be huge, containing possibly millions of pixels (variables x_j) for every layer in the network. This makes standard NNs not practical for dealing with images.

5.1 Standard convolution: image filtering

A well known operation in image processing is applying the convolution kernel. First, an image is represented in memory as a 2D array of numbers, each representing the brightness level of a pixel (see Fig)

`https://en.wikipedia.org/wiki/Kernel_(image_processing)`

`https://towardsdatascience.com/a-beginners-guide-to-convolutional-neural-networks-cnn`