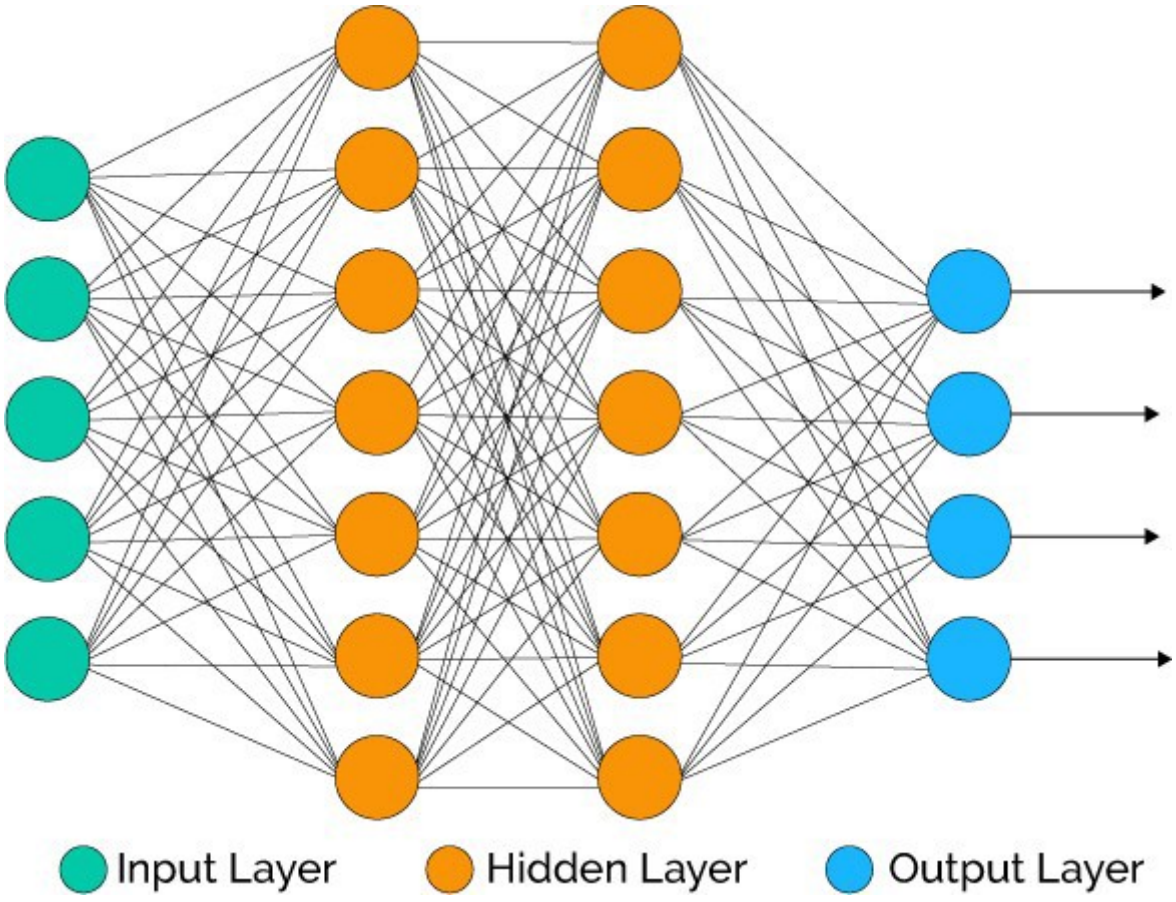
TOPIC 3

Optimality, convexity, and gradient descent

7 Neural networks

Course: Math 535 (http://www.math.wisc.edu/~roch/mmids/) - Mathematical Methods in Data Science (MMiDS)
Author: Sebastien Roch (http://www.math.wisc.edu/~roch/), Department of Mathematics, University of Wisconsin-Madison
Updated: Nov 12, 2020

Feedforward neural networks generalize the previous progressive examples with the addition of an arbitrary number of layers.



(Source) (https://towardsdatascience.com/machine-learning-fundamentals-ii-neural-networks-f1e7b2cb3eef)

7.1 Background

Each of the main layers has two components, an affine map and a nonlinear activation function. For the latter, we restrict ourselves here to the sigmoid function (https://en.wikipedia.org/wiki/Sigmoid_function) (there are many other popular choices of activation functions (https://www.tensorflow.org/api_docs/python/tf/keras/activations)), that is, the function

$$\sigma(t) = \frac{1}{1+e^{-t}} \qquad t \in \mathbb{R}.$$

Its derivative is easy to compute

$$\sigma'(t) = \frac{e^{-t}}{(1+e^{-t})^2} = \frac{1}{1+e^{-t}}\left(1 - \frac{1}{1+e^{-t}}\right) = \sigma(t)(1-\sigma(t)).$$

The latter expression is known as the logistic differential equation (https://en.wikipedia.org/wiki/Logistic_function#Logistic_differential_equation). It arises in a variety of applications, including the modeling of population dynamics (https://towardsdatascience.com/covid-19-infection-in-italy-mathematical-models-and-predictions-7784b4d7dd8d). When applied to a vector component-wise, we denote the sigmoid function by $\sigma_\odot$, similarly, $\sigma'_\odot$ for the derivative.

We consider an arbitrary number of layers $L+1$. Layer i, $i = 1, \dots, L$, is defined by a continuously differentiable function g_i which takes two vector-valued inputs: a vector of parameters $\mathbf{w}_i \in \mathbb{R}^{r_i}$ specific to the i-th layer and a vector of inputs $\mathbf{z}_i \in \mathbb{R}^{n_i}$ which is fed from the $i-1$-th layer

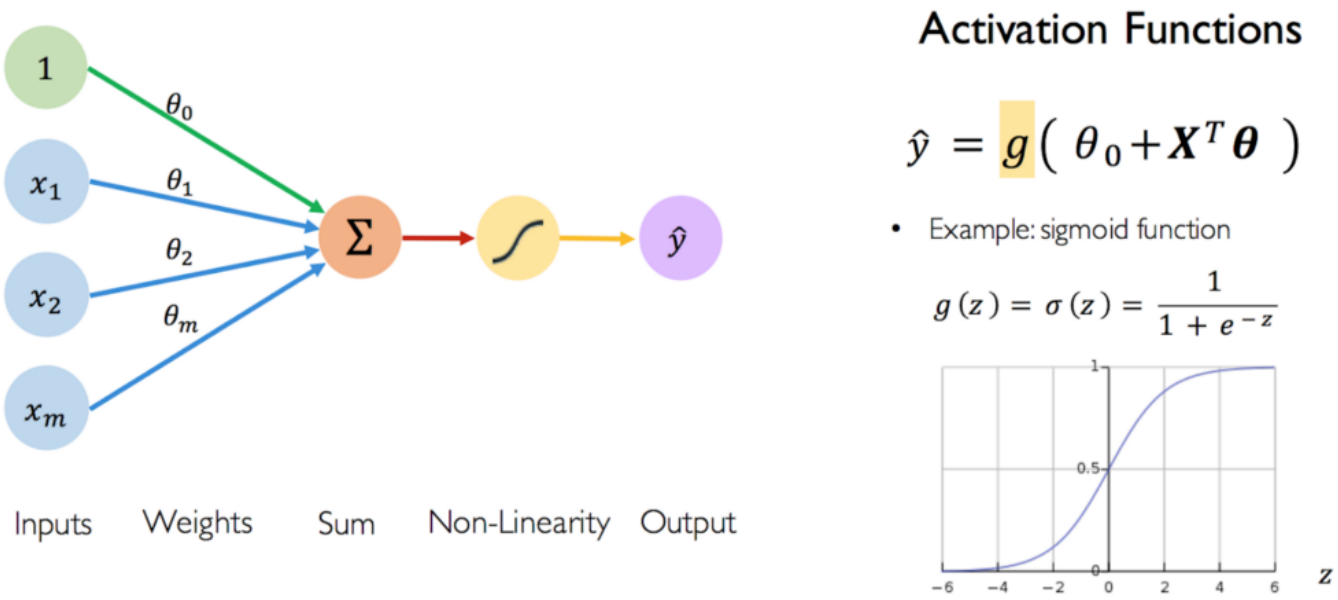
$$g_i : \mathbb{R}^{r_i+n_i} \to \mathbb{R}^{n_{i+1}}.$$

The output of g_i is a vector in $\mathbb{R}^{n_{i+1}}$ which is passed to the $i+1$-th layer as input. Here $r_i = n_{i+1}(n_i+1)$ and $\mathbf{w}_i = (\mathbf{w}_i^{(1)}; \dots; \mathbf{w}_i^{(n+i+1)})$ are the parameters with $\mathbf{w}_i^{(k)} \in \mathbb{R}^{n_i+1}$. Specifically, g_i is given by

$$g_i(\mathbf{w}_i; \mathbf{z}_i) = \left[\sigma\left(\sum_{j=1}^{n_i} w_{i,j}^{(k)} z_{i,j} + w_{i,n_i+1}^{(k)}\right)\right]_{k=1}^{n_{i+1}} = \sigma_\odot\left(\mathcal{W}_i \mathbf{z}_i^{:1}\right)$$

where we define $\mathcal{W}_i \in \mathbb{R}^{n_{i+1}\times(n_i+1)}$ as the matrix with rows $(\mathbf{w}_i^{(1)})^T, \dots, (\mathbf{w}_i^{(n_{i+1})})^T$.

Each component of g_i is referred to as a neuron.



MIT: Alexander Amini, 2018 introtodeeplearning.com

(Source) (https://hackernoon.com/hn-images/1*zjzWdMucBfRbkqMzgm2xKg.png)

The first layer takes as input $\mathbf{z}_1 = \mathbf{x} \in \mathbb{R}^d$ so that $n_1 = d$. As in logistic regression, layer L has $K-1$ outputs so that $n_{L+1} = K-1$.

Also as in logistic regression, layer $L+1$ is the modified softmax function

$$g_{L+1}(\mathbf{z}_{L+1}) = \tilde{\gamma}(\mathbf{z}_{L+1}),$$

which has no parameter.

So the output of the classifier with parameters $\mathbf{w} = (\mathbf{w}_1; \ldots; \mathbf{w}_L)$ on input $\mathbf{x}$ is

$$h_{\mathbf{x}}(\mathbf{w}) = g_{L+1}(g_L(\mathbf{w}_L; \cdots g_2(\mathbf{w}_2; g_1(\mathbf{w}_1; \mathbf{x})) \cdots)).$$

We use the cross-entropy for loss function. That is, we set

$$\ell_{\mathbf{y}}(\mathbf{z}_{L+1}) = H(\mathbf{y}, \mathbf{z}_{L+1}) = -\sum_{i=1}^{K} y_i \log z_{L+1,i}\,.$$

Finally,

$$f_{\mathbf{x},\mathbf{y}}(\mathbf{w}) = \ell_{\mathbf{y}}(h_{\mathbf{x}}(\mathbf{w})).$$

7.2 Computing the gradient

We now detail how to compute the gradient of $f_{\mathbf{x},\mathbf{y}}(\mathbf{w})$. In the forward loop, we first set $\mathbf{z}_1 := \mathbf{x}$ and then we compute for $i = 1, \ldots, L$

$$\mathbf{z}_{i+1} := g_i(\mathbf{w}_i; \mathbf{z}_i) = \left[\sigma\left(\sum_{j=1}^{n_i} w_{i,j}^{(k)} z_{i,j} + w_{i,n_i+1}^{(k)}\right)\right]_{k=1}^{n_{i+1}} = \sigma_\odot\left(\mathcal{W}_i \mathbf{z}_i^{:1}\right)$$

$$\begin{pmatrix} A_i & B_i \end{pmatrix} := J_{g_i}(\mathbf{w}_i; \mathbf{z}_i).$$

To compute the Jacobian of g_i, we use the *Chain Rule* on the composition $g_i(\mathbf{w}_i; \mathbf{z}_i) = \sigma_\odot(h_i(\mathbf{w}_i; \mathbf{z}_i))$ where we define $h_i(\mathbf{w}_i; \mathbf{z}_i) = \mathcal{W}_i \mathbf{z}_i^{:1}$. That is,

$$J_{g_i}(\mathbf{w}_i; \mathbf{z}_i) = J_{\sigma_\odot}\left(\mathcal{W}_i \mathbf{z}_i^{:1}\right) J_{h_i}(\mathbf{w}_i; \mathbf{z}_i).$$

In our analysis of logistic regression, we computed the Jacobian of h_i. We obtained

$$J_{h_i}(\mathbf{w}_i; \mathbf{z}_i) = \begin{pmatrix} \mathbb{A}_{n_i,n_{i+1}}[\mathbf{z}_i] & \mathbb{B}_{n_i,n_{i+1}}[\mathbf{w}_i] \end{pmatrix}.$$

Recall that

$$\mathbb{A}_{n_i,n_{i+1}}[\mathbf{z}_i] = \begin{pmatrix} \mathbf{e}_1 (\mathbf{z}_i^{:1})^T & \cdots & \mathbf{e}_{n_{i+1}} (\mathbf{z}_i^{:1})^T \end{pmatrix}$$

where here $\mathbf{e}_j$ is the j-th canonical basis vector in $\mathbb{R}^{n_{i+1}}$ and

$$\mathbb{B}_{n_i,n_{i+1}}[\mathbf{w}_i] = \begin{pmatrix} ((\mathbf{w}_i^{(1)})^{:n_i})^T \\ \vdots \\ ((\mathbf{w}_i^{(n_{i+1})})^{:n_i})^T \end{pmatrix}.$$

The Jacobian of $\sigma_\odot(\mathbf{t})$ can be computed from σ'. Indeed,

$$\frac{\partial}{\partial t_j}\sigma_\odot(\mathbf{t})_j = \frac{\partial}{\partial t_j}\sigma(t_j) = \sigma'(t_j) = \frac{e^{-t_j}}{(1+e^{-t_j})^2},$$

while for $\ell \neq j$

$$\frac{\partial}{\partial t_j}\sigma_\odot(\mathbf{t})_\ell = \frac{\partial}{\partial t_j}\sigma(t_\ell) = 0.$$

In other words, the j-th column of the Jacobian is $\sigma'(t_j)\,\mathbf{e}_j$ where again $\mathbf{e}_j$ is the j-th canonical basis vector in $\mathbb{R}^{n_{i+1}}$. So $\boldsymbol{J}_{\sigma_\odot}(\mathbf{t})$ is the diagonal matrix with diagonal entries $\sigma'(t_j)$, $j = 1, \ldots, n_{i+1}$, which we denote

$$\boldsymbol{J}_{\sigma_\odot}(\mathbf{t}) = \operatorname{diag}(\sigma'_\odot(\mathbf{t})).$$

Combining the previous formulas, we get

$$\begin{aligned}\boldsymbol{J}_{g_i}(\mathbf{w}_i;\mathbf{z}_i) &= \boldsymbol{J}_{\sigma_\odot}\left(\mathcal{W}_i\mathbf{z}_i^{:1}\right)\boldsymbol{J}_{h_i}(\mathbf{w}_i;\mathbf{z}_i)\\ &= \operatorname{diag}\left(\sigma'_\odot\left(\mathcal{W}_i\mathbf{z}_i^{:1}\right)\right)\begin{pmatrix}\mathbb{A}_{n_i,n_{i+1}}[\mathbf{z}_i] & \mathbb{B}_{n_i,n_{i+1}}[\mathbf{w}_i]\end{pmatrix}\\ &= \begin{pmatrix}\widetilde{\mathbb{A}}_{n_i,n_{i+1}}[\mathbf{w}_i;\mathbf{z}_i] & \widetilde{\mathbb{B}}_{n_i,n_{i+1}}[\mathbf{w}_i;\mathbf{z}_i]\end{pmatrix}\end{aligned}$$

where we define

$$\widetilde{\mathbb{A}}_{n_i,n_{i+1}}[\mathbf{w}_i;\mathbf{z}_i] = \begin{pmatrix}\sigma'\left((\mathbf{w}_i^{(1)})^T\,\mathbf{z}_i^{:1}\right)\mathbf{e}_1(\mathbf{z}_i^{:1})^T & \cdots & \sigma'\left((\mathbf{w}_i^{(n_{i+1})})^T\,\mathbf{z}_i^{:1}\right)\mathbf{e}_{n_{i+1}}(\mathbf{z}_i^{:1})^T\end{pmatrix}$$

and

$$\widetilde{\mathbb{B}}_{n_i,n_{i+1}}[\mathbf{w}_i;\mathbf{z}_i] = \begin{pmatrix}\sigma'\left((\mathbf{w}_i^{(1)})^T\,\mathbf{z}_i^{:1}\right)((\mathbf{w}_i^{(1)})^{:n_i})^T\\ \vdots\\ \sigma'\left((\mathbf{w}_i^{(n_{i+1})})^T\,\mathbf{z}_i^{:1}\right)((\mathbf{w}_i^{(n_{i+1})})^{:n_i})^T\end{pmatrix}.$$

For layer $L+1$, we have previously computed the Jacobian of the modified softmax function. We get

$$\begin{aligned}\mathbf{z}_{L+2} &:= g_{L+1}(\mathbf{z}_{L+1}) = \tilde{\gamma}(\mathbf{z}_{L+1})\\ B_{L+1} &:= \boldsymbol{J}_{g_{L+1}}(\mathbf{z}_{L+1}) = \boldsymbol{J}_{\tilde{\gamma}}(\mathbf{z}_{L+1}) = \mathbb{C}_K[\mathbf{z}_{L+1}].\end{aligned}$$

Refer to the previous notebook for an explicit formula for $\mathbb{C}_K[\mathbf{z}_{L+1}]$.

Also, as in the multinomial logistic regression case, the loss and gradient of the loss are

$$\begin{aligned}\mathbf{z}_{L+3} &:= \ell_{\mathbf{y}}(\mathbf{z}_{L+2}) = -\sum_{i=1}^{K} y_i \log z_{L+2,i}\\ \mathbf{q}_{L+2} &:= \nabla\ell_{\mathbf{y}}(\mathbf{z}_{L+2}) = \left[-\frac{y_i}{z_{L+2,i}}\right]_{i=1}^{K}.\end{aligned}$$

We summarize the whole procedure next. We use $\mathbf{v}_1 \odot \mathbf{v}_2$ to denote the component-wise product of the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$.

Initialization: $\mathbf{z}_1 := \mathbf{x}$

Forward layer loop: For $i = 1, \dots, L$:

$$\begin{aligned} \mathbf{z}^{i+1} &:= g_i(\mathbf{w}^i; \mathbf{z}^i) = \sigma_\odot\left(\mathcal{W}_i \mathbf{z}_i^{:1}\right) \\ \begin{pmatrix} A_i & B_i \end{pmatrix} &:= J_{g_i}(\mathbf{w}^i; \mathbf{z}^i) = \begin{pmatrix} \widetilde{\mathbb{A}}_{n_i,n_{i+1}}[\mathbf{w}_i; \mathbf{z}_i] & \widetilde{\mathbb{B}}_{n_i,n_{i+1}}[\mathbf{w}_i; \mathbf{z}_i] \end{pmatrix} \end{aligned}$$

and

$$\begin{aligned} \mathbf{z}_{L+2} &:= g_{L+1}(\mathbf{z}_{L+1}) = \tilde{\gamma}(\mathbf{z}_{L+1}) \\ B_{L+1} &:= J_{g_{L+1}}(\mathbf{z}_{L+1}) = \mathbb{C}_K[\mathbf{z}_{L+1}]. \end{aligned}$$

Loss:

$$\begin{aligned} \mathbf{z}_{L+3} &:= \ell_{\mathbf{y}}(\mathbf{z}^{L+2}) \\ \mathbf{q}_{L+2} &:= \nabla \ell_{\mathbf{y}}(\mathbf{z}^{L+2}) = \left[-\frac{y_i}{z_{L+2,i}}\right]_{i=1}^{K}. \end{aligned}$$

Backward layer loop:

$$\mathbf{q}_{L+1} := B_{L+1}^T \mathbf{q}_{L+2} = \mathbb{C}_K[\mathbf{z}_{L+1}]^T \mathbf{q}_{L+2}$$

and for $i = L, \dots, 1$:

$$\begin{aligned} \mathbf{p}_i &:= A_i^T \mathbf{q}_{i+1} \\ &= \widetilde{\mathbb{A}}_{n_i,n_{i+1}}[\mathbf{w}_i; \mathbf{z}_i]^T \mathbf{q}_{i+1} \\ &= \left(\sigma'\left((\mathbf{w}_i^{(1)})^T \mathbf{z}_i^{:1}\right) \mathbf{z}_i^{:1}; \cdots; \sigma'\left((\mathbf{w}_i^{(n_{i+1})})^T \mathbf{z}_i^{:1}\right) \mathbf{z}_i^{:1}\right) \odot \mathbf{q}_{i+1} \\ \mathbf{q}_i &:= B_i^T \mathbf{q}_{i+1} = \widetilde{\mathbb{B}}_{n_i,n_{i+1}}[\mathbf{w}_i; \mathbf{z}_i]^T \mathbf{q}_{i+1} \end{aligned}$$

Output:

$$\nabla f_{\mathbf{x},\mathbf{y}}(\mathbf{w}) = (\mathbf{p}_L; \dots; \mathbf{p}_1).$$

7.3 Implementation

We implement a neural network in Flux. We use the MNIST dataset again. We first load the data and convert it to an appropriate matrix representation. Recall that the data can be accessed with `Flux.Data.MNIST`.

In [1]:
```
#Julia version: 1.5.1
ENV["JULIA_CUDA_SILENT"] = true # silences warning about GPUs

using Statistics, Images, QuartzImageIO
using Flux, Flux.Data.MNIST, Flux.Data.FashionMNIST
using Flux: mse, train!, Data.DataLoader, throttle
using Flux: onehot, onehotbatch, onecold, crossentropy
using IterTools: ncycle
```

In [2]:
```
imgs = MNIST.images()
labels = MNIST.labels()
Xtrain = reduce(hcat, [reshape(Float32.(imgs[i]),:) for i = 1:length(img
s)]);
ytrain = onehotbatch(labels, 0:9);
```

In [3]:
```
test_imgs = MNIST.images(:test)
test_labels = MNIST.labels(:test)
Xtest = reduce(hcat,
    [reshape(Float32.(test_imgs[i]),:) for i = 1:length(test_imgs)])
ytest = onehotbatch(test_labels, 0:9);
```

We use `Chain` to construct a three-layer model.

In [4]:
```
m = Chain(
  Dense(28^2, 32, σ),
  Dense(32, 10),
  softmax
)
```

Out[4]:
```
Chain(Dense(784, 32, σ), Dense(32, 10), softmax)
```

In [5]:
```
accuracy(x, y) = mean(onecold(m(x), 0:9) .== onecold(y, 0:9));
```

In [6]:
```
accuracy(Xtest, ytest)
```

Out[6]:
```
0.1247
```

We then proceed as before.

In [7]:
```
loader = DataLoader(Xtrain, ytrain; batchsize=128, shuffle=true)
accuracy(x, y) = mean(onecold(m(x), 0:9) .== onecold(y, 0:9))
loss(x, y) = crossentropy(m(x), y)
ps = params(m)
opt = ADAM()
evalcb = () -> @show(accuracy(Xtest,ytest));
```

In [8]:
```
@time train!(loss, ps, ncycle(loader, Int(2e1)), opt, cb = throttle(eval
cb, 2))
```

```
accuracy(Xtest, ytest) = 0.1265
accuracy(Xtest, ytest) = 0.9416
accuracy(Xtest, ytest) = 0.9529
accuracy(Xtest, ytest) = 0.9605
 25.973718 seconds (52.72 M allocations: 13.555 GiB, 2.89% gc time)
```

The final accuracy is better than that achieved with multiclass logistic regression:

In [9]:
```
accuracy(Xtest, ytest)
```

Out[9]: 0.9613

As before, to make a prediction, we use `m(x)` and `onecold`.

In [10]:
```
onecold(m(Xtest[:,1]), 0:9)
```

Out[10]: 7

In [11]:
```
onecold(ytest[:,1], 0:9)
```

Out[11]: 7