TOPIC 3 ¶

Optimality, convexity, and gradient descent

6 Automatic differentiation: examples

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

We give several concrete examples of progressive functions and of the application of the algorithmic differentiation method from the previous section.

6.1 Example 1: linear regression

We begin with linear regression.

6.1.1 Computing the gradient

While we have motivated the framework introduced the previous section from the point of view of classification, it also immediately applies to the regression setting.

Here y is a real-valued outcome variable. We revisit the case of linear regression where the loss function is

$$\ell_y(z_2) = (z_2 - y)^2$$

and the regression function has a single layer (that is, $L = 1$) with

$$h_{\mathbf{x}}(\mathbf{w}) = g_1(\mathbf{w}_1; \mathbf{z}_1) = \sum_{j=1}^{d} w_{1,j} z_{1,j} + w_{1,d+1}$$

where $\mathbf{w}_1 = \mathbf{w} \in \mathbb{R}^{d+1}$ are the parameters and $\mathbf{z}_1 = \mathbf{x} \in \mathbb{R}^d$ is the input. Hence,

$$f_{\mathbf{x},y}(\mathbf{w}) = \ell_y(g_1(\mathbf{w}_1; \mathbf{x})) = \left(\sum_{j=1}^{d} w_{1,j} x_j + w_{1,d+1} - y \right)^2.$$

It will convenient to introduce the following notation. For a vector $\mathbf{z} \in \mathbb{R}^d$, the vector $\mathbf{z}^{:1} \in \mathbb{R}^{d+1}$ concatenates a 1 at the end of $\mathbf{z}$, that is, $\mathbf{z}^{:1} = (\mathbf{z}; 1)$. For a vector $\mathbf{w} \in \mathbb{R}^{d+1}$, the vector $\mathbf{w}^{:d} \in \mathbb{R}^d$ drops the last entry of $\mathbf{w}$, that is, $\mathbf{w}^{:d} = (w_1, \ldots, w_d)$.

The forward pass in this case is:

Initialization:

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

Forward layer loop:

$$z_2 := g_1(\mathbf{w}_1; \mathbf{z}_1) = \mathbf{w}_1^T \mathbf{z}_1^{:1}$$
$$\begin{pmatrix} A_1 & B_1 \end{pmatrix} := J_{g_1}(\mathbf{w}_1; \mathbf{z}_1) = (\mathbf{z}_1^{:1}; \mathbf{w}_1^{:d})^T$$

Loss:

$$z_3 := \ell_y(z_2) = (z_2 - y)^2 = (\mathbf{w}^T \mathbf{x}^{:1} - y)^2$$
$$q_2 := \frac{\mathrm{d}}{\mathrm{d}z_2} \ell_y(z_2) = 2(z_2 - y).$$

The backward pass is:

Backward layer loop:

$$\mathbf{p}_1 := A_1^T q_2 = 2(z_2 - y)\, \mathbf{z}_1^{:1}$$

Output:

$$\nabla f_{\mathbf{x},y}(\mathbf{w}) = \mathbf{p}_1 = 2 \left(\mathbf{w}^T \mathbf{x}^{:1} - y \right) \mathbf{x}^{:1}.$$

There is in fact no need to compute B_1 (and $\mathbf{q}_1$).

When applied to a mini-batch of samples $B \subseteq [n]$, we compute the average of the gradients over the samples in B, that is,

$$\frac{1}{|B|} \sum_{i \in B} \nabla f_{\mathbf{x}_i, y_i}(\mathbf{w}) = \frac{2}{|B|} \sum_{i \in B} \left(\mathbf{w}^T \mathbf{x}_i^{:1} - y_i \right) \mathbf{x}_i^{:1}.$$

6.1.2 Flux

We will be using Flux.jl (https://fluxml.ai) to implement the previous method. The `Dense` function constructs an affine map from `in` predictor variables to `out` response variables. It is defined a special way that its parameters, the matrix `W` and the intercept vector `b`, can be accessed by `m.W` and `m.b` if `m` is the name given to the function.

We will also use some other utility functions. The function `mse` computes the mean squared error (MSE). It will be our loss function in this section. The functions `DataLoader` and `ncycle` allow us to construct mini-batches in a straightforward way. Finally `thottle` is used to print progress messages.

The help rubric of each of these is below.

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

using CSV, DataFrames, GLM, 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]:
```
?Dense
```

```
search: Dense DenseArray DenseVector DenseMatrix DenseVecOrMat DenseConvDims
```

Out[2]:
```
Dense(in::Integer, out::Integer, σ = identity)
```

Create a traditional `Dense` layer with parameters `W` and `b`.

```
y = σ.(W * x .+ b)
```

The input `x` must be a vector of length `in`, or a batch of vectors represented as an `in × N` matrix. The out `y` will be a vector or batch of length `out`.

Example

```
julia> d = Dense(5, 2)
Dense(5, 2)

julia> d(rand(5))
2-element Array{Float32,1}:
 -0.16210233
  0.123119034
```

In [3]:
```
?mse
```

```
search: mse rmse imstretch imresize SumSquaredDifference CompositeExcep
tion
```

Out[3]:

```
mse(ŷ, y; agg=mean)
```

Return the loss corresponding to mean square error:

```
agg((ŷ .- y).^2)
```

In [4]:
```
?DataLoader
```

search:

Out[4]: `DataLoader(data; batchsize=1, shuffle=false, partial=true)`

An object that iterates over mini-batches of `data`, each mini-batch containing `batchsize` observations (except possibly the last one).

Takes as input a single data tensor, or a tuple (or a named tuple) of tensors. The last dimension in each tensor is considered to be the observation dimension.

If `shuffle=true`, shuffles the observations each time iterations are re-started. If `partial=false`, drops the last mini-batch if it is smaller than the batchsize.

The original data is preserved in the `data` field of the DataLoader.

Usage example:

```
Xtrain = rand(10, 100)
train_loader = DataLoader(Xtrain, batchsize=2)
# iterate over 50 mini-batches of size 2
for x in train_loader
    @assert size(x) == (10, 2)
    ...
end

train_loader.data   # original dataset

# similar, but yielding tuples
train_loader = DataLoader((Xtrain,), batchsize=2)
for (x,) in train_loader
    @assert size(x) == (10, 2)
    ...
end

Xtrain = rand(10, 100)
Ytrain = rand(100)
train_loader = DataLoader((Xtrain, Ytrain), batchsize=2, shuffle=true)
for epoch in 1:100
    for (x, y) in train_loader
        @assert size(x) == (10, 2)
        @assert size(y) == (2,)
        ...
    end
end

# train for 10 epochs
using IterTools: ncycle
Flux.train!(loss, ps, ncycle(train_loader, 10), opt)

# can use NamedTuple to name tensors
train_loader = DataLoader((images=Xtrain, labels=Ytrain), batchsize=2,
 shuffle=true)
for datum in train_loader
    @assert size(datum.images) == (10, 2)
    @assert size(datum.labels) == (2,)
end
```

In [5]:
```
?ncycle
```

```
search:
```

Out[5]:
```
ncycle(iter, n)
```

Cycle through `iter` `n` times.

```
jldoctest
julia> for i in ncycle(1:3, 2)
           @show i
       end
i = 1
i = 2
i = 3
i = 1
i = 2
i = 3
```

In [6]:
```
?throttle
```

```
search:
```

Out[6]:
```
throttle(f, timeout; leading=true, trailing=false)
```

Return a function that when invoked, will only be triggered at most once during `timeout` seconds.

Normally, the throttled function will run as much as it can, without ever going more than once per `wait` duration; but if you'd like to disable the execution on the leading edge, pass `leading=false` . To enable execution on the trailing edge, pass `trailing=true` .

6.1.3 The `Advertising` dataset and the least-squares solution

We return to the `Advertising` dataset.

In [7]:
```
df = DataFrame(CSV.File("advertising.csv"))
first(df,5)
```

Out[7]: 5 rows × 5 columns

	Column1	TV	radio	newspaper	sales
	Int64	Float64	Float64	Float64	Float64
1	1	230.1	37.8	69.2	22.1
2	2	44.5	39.3	45.1	10.4
3	3	17.2	45.9	69.3	9.3
4	4	151.5	41.3	58.5	18.5
5	5	180.8	10.8	58.4	12.9

In [8]:
```
n = nrow(df)
```

Out[8]:
```
200
```

We first compute the solution using the least-squares approach we detailed previously.

In [9]:
```
X = reduce(hcat, [df[:,:TV], df[:,:radio], df[:,:newspaper]])
Xaug = hcat(ones(n), X)
y = df[:,:sales];
```

In [10]:
```
@time q = Xaug\y
```

```
  0.873739 seconds (2.89 M allocations: 141.946 MiB, 2.85% gc time)
```

Out[10]:
```
4-element Array{Float64,1}:
  2.938889369459415
  0.04576464545539759
  0.18853001691820445
 -0.0010374930424763011
```

The MSE is:

In [11]:
```
mean((Xaug*q .- y).^2)
```

Out[11]:
```
2.7841263145109356
```

6.1.4 Solving the problem using Flux

We use `DataLoader` to set up the data for Flux. Note that it takes the transpose of what we have been using, that is, the columns of the data matrix correspond to the samples. Here we take mini-batches of size `batchsize=20` and the option `shuffle=true` indicates that we apply a random permutation of the samples on every pass through the data.

In [12]:
```
Xtrain = X'
ytrain = reshape(y, (1,length(y)))
loader = DataLoader(Xtrain, ytrain; batchsize=64, shuffle=true);
```

For example, the first component of the first item is the features for the first 64 samples (after random permutation).

In [13]:
```
first(loader)[1]
```

Out[13]:
```
3×64 Array{Float64,2}:
  7.8  238.2   4.1  31.5  225.8  273.7  …  232.1  206.9  265.2  193.7
74.7
 38.9   34.3  11.6  24.6    8.2   28.9       8.6    8.4    2.9   35.4
49.4
 50.6    5.3   5.7   2.2   56.5   59.7       8.7   26.4   43.0   75.6
45.7
```

Now we construct our model. It is simply an affine map from $\mathbb{R}^3$ to $\mathbb{R}$.

In [14]:
```
m = Dense(3, 1)
```

Out[14]:
```
Dense(3, 1)
```

The loss function is the MSE.

In [15]:
```
loss(x,y) = mse(m(x),y)
```

Out[15]:
```
loss (generic function with 1 method)
```

Finally, the function `train!` (https://fluxml.ai/Flux.jl/stable/training/training/#Flux.Optimise.train!) runs an optimization method of our choice on the loss function. The `!` in the function name indicates that it modifies the parameters we pass to it, in this case `m.W` and `m.b`. There are many optimizers (https://fluxml.ai/Flux.jl/stable/training/optimisers/#Optimiser-Reference-1) available . Stochastic gradient descent can chosen with Descent (https://fluxml.ai/Flux.jl/stable/training/optimisers/#Flux.Optimise.Descent). (But it is slow. Instead we will use the popular ADAM (https://fluxml.ai/Flux.jl/stable/training/optimisers/#Flux.Optimise.ADAM). See this post (https://hackernoon.com/demystifying-different-variants-of-gradient-descent-optimization-algorithm-19ae9ba2e9bc) for a brief explanation of many common optimizers.)

We also pass the parameters to `train!` using `params` and a callback function `evalcb()` that prints progress.

Choosing the right number of passes (i.e. epochs) through the data requires some experimenting. Here 10^4 suffices.

In [16]:
```
ps = params(m)
opt = Descent(1e-5)
evalcb =() -> @show(loss(Xtrain,ytrain));
```

In [17]:
```
@time train!(loss, ps, ncycle(loader, Int(1e4)), opt, cb = throttle(eval
cb, 2))
```

```
loss(Xtrain, ytrain) = 2160.6028562802303
loss(Xtrain, ytrain) = 3.8571577593201845
 18.668849 seconds (75.40 M allocations: 3.075 GiB, 4.78% gc time)
```

The final parameters and loss are:

In [18]:
```
m.b
```

Out[18]:
```
1-element Array{Float32,1}:
 0.3245267
```

In [19]:
```
m.W
```

Out[19]:
```
1×3 Array{Float32,2}:
 0.0509482  0.218272  0.0144676
```

In [20]:
```
loss(Xtrain,ytrain)
```

Out[20]:
```
3.9033906662010316
```

6.2 Example 2: multinomial logistic regression

We return to classification. We first appeal to multinomial logistic regression (https://en.wikipedia.org/wiki/Multinomial_logistic_regression) to learn a classifier over K labels. Recall that we encode label i as the K-dimensional vector $\mathbf{e}_i$ and that we allow the output of the classifier to be a probability distribution over the labels $\{1, \ldots, K\}$. Observe that $\mathbf{e}_i$ can itself be thought of as a probability distribution, one that assigns probability one to i.

6.2.1 Background

In multinomial logistic regression, we once again use an affine function of the input data.

This time, we have K functions that output a score associated to each label. We then transform these scores into a probability distribution over the K labels. There are many ways of doing this. A standard approach is the softmax function (https://en.wikipedia.org/wiki/Softmax_function): for $\mathbf{z} \in \mathbb{R}^K$

$$\gamma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}, \quad i = 1, \dots, K.$$

To explain the name, observe that the larger inputs are mapped to larger probabilities.

In fact, since a probability distribution must to 1, it is determined by the probabilities assigned to the first $K-1$ labels. In other words, we can drop the score associated to the last label. Formally, we use the modifed softmax function

$$\tilde{\gamma}(\mathbf{z})_i = \begin{cases} \frac{e^{z_i}}{1+\sum_{j=1}^{K-1} e^{z_j}} & i = 1, \dots, K-1 \\ \frac{1}{1+\sum_{j=1}^{K-1} e^{z_j}} & i = K \end{cases}$$

where this time $\mathbf{z} \in \mathbb{R}^{K-1}$.

Hence we now have two layers, that is, $L = 2$. The layers are defined by the functions

$$z_{2,k} = g_1(\mathbf{w}_1; \mathbf{z}_1)_k = \sum_{j=1}^d w_{1,j}^{(k)} z_{1,j} + w_{1,d+1}^{(k)}, \quad k = 1, \dots, K-1$$

where $\mathbf{w}_1 = (\mathbf{w}_1^{(1)}; \dots; \mathbf{w}_1^{(K-1)})$ are the parameters with $\mathbf{w}_1^{(k)} \in \mathbb{R}^{d+1}$ and $\mathbf{z}_1 = \mathbf{x} \in \mathbb{R}^d$ is the input, and

$$\mathbf{z}_3 = g_2(\mathbf{z}_2) = \tilde{\gamma}(\mathbf{z}_2),$$

where $\tilde{\gamma}$ is the modified softmax function. Note that the latter has no associated parameter.

So the output of the classifier with parameters $\mathbf{w} = \mathbf{w}_1 = (\mathbf{w}_1^{(1)}; \dots; \mathbf{w}_1^{(K-1)})$ on input $\mathbf{x}$ is

$$\begin{aligned} h_{\mathbf{x}}(\mathbf{w})_i &= g_2(g_1(\mathbf{w}_1; \mathbf{z}_1))_i \\ &= \tilde{\gamma}\left(\left[\sum_{j=1}^d w_{1,j}^{(k)} x_j + w_{1,d+1}^{(k)} \right]_{k=1}^{K-1} \right)_i, \end{aligned}$$

for $i = 1, \dots, K$.

It remains to define a loss function. To quantify the fit, it is natural to use a notion of distance between probability measures, here between the output $h_{\mathbf{x}}(\mathbf{w}) \in \Delta_K$ and the correct label $\mathbf{y} \in \{\mathbf{e}_1, \ldots, \mathbf{e}_K\} \subseteq \Delta_K$. There are many such measures. In multinomial logistic regression, we use the Kullback-Leibler divergence (https://en.wikipedia.org/wiki/Kullback–Leibler_divergence). For two probability distributions $\mathbf{p}, \mathbf{q} \in \Delta_K$, it is defined as

$$\mathrm{KL}(\mathbf{p}\|\mathbf{q}) = \sum_{i=1}^{K} p_i \log \frac{p_i}{q_i}$$

where it will suffice to restrict ourselves to the case $\mathbf{q} > \mathbf{0}$ and where we use the convention $0 \log 0 = 0$ (so that terms with $p_i = 0$ contribute 0 to the sum).

Exercise: Show that $\log x \leq x - 1$ for all $x > 0$. [Hint: Compute the derivative of $s(x) = x - 1 - \log x$ and the value $s(1)$.] ◁

Notice that $\mathbf{p} = \mathbf{q}$ implies $\mathrm{KL}(\mathbf{p}\|\mathbf{q}) = 0$. We show that $\mathrm{KL}(\mathbf{p}\|\mathbf{q}) \geq 0$, a result known as *Gibbs' inequality*.

Theorem (Gibbs): For any $\mathbf{p}, \mathbf{q} \in \Delta_K$ with $\mathbf{q} > \mathbf{0}$,

$$\mathrm{KL}(\mathbf{p}\|\mathbf{q}) \geq 0.$$

Proof: Let I be the set of indices i such that $p_i > 0$. Hence

$$\mathrm{KL}(\mathbf{p}\|\mathbf{q}) = \sum_{i \in I} p_i \log \frac{p_i}{q_i}.$$

Using the exercise above, $\log x \leq x - 1$ for all $x > 0$ so that

$$\begin{aligned}
\mathrm{KL}(\mathbf{p}\|\mathbf{q}) &= -\sum_{i \in I} p_i \log \frac{q_i}{p_i} \\
&\geq -\sum_{i \in I} p_i \left(\frac{q_i}{p_i} - 1\right) \\
&= -\sum_{i \in I} q_i + \sum_{i \in I} p_i \\
&= -\sum_{i \in I} q_i + 1 \\
&\geq 0
\end{aligned}$$

where we used that $\log z^{-1} = -\log z$ on the first line and the fact that $p_i = 0$ for all $i \notin I$ on the fourth line. □

Going back to the loss function, we use the identity $\log \frac{\alpha}{\beta} = \log \alpha - \log \beta$ to re-write

$$\begin{aligned}\mathrm{KL}(\mathbf{y} \| h_{\mathbf{x}}(\mathbf{w})) &= \sum_{i=1}^{K} y_i \log \frac{y_i}{h_{\mathbf{x}}(\mathbf{w})_i} \\ &= \sum_{i=1}^{K} y_i \log y_i - \sum_{i=1}^{K} y_i \log h_{\mathbf{x}}(\mathbf{w})_i .\end{aligned}$$

Notice that the first term on right-hand side does not depend on $\mathbf{w}$. Hence we can ignore when optimizing $\mathrm{KL}(\mathbf{y} \| h_{\mathbf{x}}(\mathbf{w}))$. The remaining term

$$H(\mathbf{y}, h_{\mathbf{x}}(\mathbf{w})) = -\sum_{i=1}^{K} y_i \log h_{\mathbf{x}}(\mathbf{w})_i$$

is known as the cross-entropy (https://en.wikipedia.org/wiki/Cross_entropy). We use it to define our loss function. That is, we set

$$\ell_{\mathbf{y}}(\mathbf{z}_3) = H(\mathbf{y}, \mathbf{z}_3) = -\sum_{i=1}^{K} y_i \log z_{3,i} .$$

Finally,

$$f_{\mathbf{x},\mathbf{y}}(\mathbf{w}) = \ell_{\mathbf{y}}(h_{\mathbf{x}}(\mathbf{w})) = -\sum_{i=1}^{K} y_i \log \tilde{\gamma}\left(\left[\sum_{j=1}^{d} w_{1,j}^{(k)} x_j + w_{1,d+1}^{(k)} \right]_{k=1}^{K-1} \right)_i .$$

6.2.2 Computing the gradient

The forward pass starts with the initialization $\mathbf{z}_1 := \mathbf{x}$.

The forward layer loop has two steps. First we compute

$$\mathbf{z}_2 := g_1(\mathbf{w}_1; \mathbf{z}_1) = \left[\sum_{j=1}^{d} w_{1,j}^{(k)} z_{1,j} + w_{1,d+1}^{(k)} \right]_{k=1}^{K-1} = \mathcal{W}_1 \mathbf{z}_1^{:1}$$

$$\begin{pmatrix} A_1 & B_1 \end{pmatrix} := J_{g_1}(\mathbf{w}_1; \mathbf{z}_1)$$

where we define $\mathcal{W} = \mathcal{W}_1 \in \mathbb{R}^{(K-1)\times(d+1)}$ as the matrix with rows $(\mathbf{w}_1^{(1)})^T, \dots, (\mathbf{w}_1^{(K-1)})^T$. To compute the Jacobian, let us look at the columns corresponding to the variables in $\mathbf{w}_1^{(k)}$, that is, columns $\alpha_k = (k-1)(d+1)+1$ to $\beta_k = k(d+1)$. Note that only component k of g_1 depends on $\mathbf{w}_1^{(k)}$, so the rows $\neq k$ of J_{g_1} are 0 for those columns. Row k on the other hand is $(\mathbf{z}_1^{:1})^T$. Hence one way to write the columns α_k to β_k of J_{g_1} is $\mathbf{e}_k(\mathbf{z}_1^{:1})^T$, where here $\mathbf{e}_k \in \mathbb{R}^{K-1}$ is the canonical basis of $\mathbb{R}^{K-1}$ (in a slight abuse of notation). So A_1 can be written in block form as

$$A_1 = \begin{pmatrix} \mathbf{e}_1(\mathbf{z}_1^{:1})^T & \cdots & \mathbf{e}_{K-1}(\mathbf{z}_1^{:1})^T \end{pmatrix} =: \mathbb{A}_{d,K-1}[\mathbf{z}_1],$$

where the last equality is a definition.

As for the columns corresponding to the variables in $\mathbf{z}_1$, that is, columns $(K-1)(d+1)+1$ to $(K-1)(d+1)+d$, each row takes the same form. Indeed row k is $((\mathbf{w}_1^{(k)})^{:d})^T$. So B_1 can be written as

$$B_1 = \begin{pmatrix} ((\mathbf{w}_1^{(1)})^{:d})^T \\ \vdots \\ ((\mathbf{w}_1^{(K-1)})^{:d})^T \end{pmatrix} =: \mathbb{B}_{d,K-1}[\mathbf{w}_1].$$

In fact, we will not need B_1 here, but we will need it in a later section.

In the second step of the forward layer loop, we compute

$$\mathbf{z}_3 := g_2(\mathbf{z}_2) = \tilde{\gamma}(\mathbf{z}_2)$$
$$B_2 := J_{g_2}(\mathbf{z}_2) = J_{\tilde{\gamma}}(\mathbf{z}_2).$$

So we need to compute the Jacobian of $\tilde{\gamma}$. We divide the computation into several cases.

When $1 \leq i = j \leq K-1$,

$$\begin{aligned}(B_2)_{ii} &= \frac{\partial}{\partial z_{2,i}}[\tilde{\gamma}(\mathbf{z}_2)_i] \\ &= \frac{\partial}{\partial z_{2,i}}\left[\frac{e^{z_{2,i}}}{1+\sum_{k=1}^{K-1} e^{z_{2,k}}}\right] \\ &= \frac{e^{z_{2,i}}\left(1+\sum_{k=1}^{K-1} e^{z_{2,k}}\right) - e^{z_{2,i}}\left(e^{z_{2,i}}\right)}{\left(1+\sum_{k=1}^{K-1} e^{z_{2,k}}\right)^2} \\ &= \frac{e^{z_{2,i}}\left(1+\sum_{k=1,k\neq i}^{K-1} e^{z_{2,k}}\right)}{\left(1+\sum_{k=1}^{K-1} e^{z_{2,k}}\right)^2} =: \mathbb{C}_K[\mathbf{z}_2]_{ii},\end{aligned}$$

by the quotient rule (https://en.wikipedia.org/wiki/Quotient_rule).

When $1 \leq i, j \leq K-1$ with $i \neq j$,

$$\begin{aligned}(B_2)_{ij} &= \frac{\partial}{\partial z_{2,j}}[\tilde{\gamma}(\mathbf{z}_2)_i] \\ &= \frac{\partial}{\partial z_{2,j}}\left[\frac{e^{z_{2,i}}}{1+\sum_{k=1}^{K-1} e^{z_{2,k}}}\right] \\ &= \frac{-e^{z_{2,i}}\left(e^{z_{2,j}}\right)}{\left(1+\sum_{k=1}^{K-1} e^{z_{2,k}}\right)^2} =: \mathbb{C}_K[\mathbf{z}_2]_{ij}.\end{aligned}$$

When $i = K$ and $1 \leq j \leq K-1$,

$$\begin{aligned}(B_2)_{ij} &= \frac{\partial}{\partial z_{2,j}}[\tilde{\gamma}(\mathbf{z}_2)_i] \\ &= \frac{\partial}{\partial z_{2,j}}\left[\frac{1}{1+\sum_{k=1}^{K-1} e^{z_{2,k}}}\right] \\ &= \frac{-\left(e^{z_{2,j}}\right)}{\left(1+\sum_{k=1}^{K-1} e^{z_{2,k}}\right)^2} =: \mathbb{C}_K[\mathbf{z}_2]_{ij}.\end{aligned}$$

The gradient of the loss function is

$$\mathbf{q}_3 = \nabla_{\ell_y}(\mathbf{z}_3) = \left[-\frac{y_i}{z_{3,i}}\right]_{i=1}^{K}.$$

The backward layer loop also has two steps. Because g_2 does not have parameters, we only need to compute $\mathbf{q}_2$ and $\mathbf{p}_1$. We get for $1 \leq j \leq K-1$

$$\begin{aligned}
q_{2,j} = [B_2^T \mathbf{q}_3]_j &= \sum_{i=1}^{K} \mathbb{C}_{d,K}[\mathbf{z}_2]_{ij} \left(-\frac{y_i}{\tilde{\gamma}(\mathbf{z}_2)_i} \right) \\
&= -\frac{y_j \left(1 + \sum_{k=1,k\neq j}^{K-1} e^{z_{2,k}}\right)}{1 + \sum_{k=1}^{K-1} e^{z_{2,k}}} + \sum_{k=1,k\neq j}^{K} \frac{y_k e^{z_{2,j}}}{1 + \sum_{k=1}^{K-1} e^{z_{2,k}}} \\
&= -\frac{y_j \left(1 + \sum_{k=1,k\neq j}^{K-1} e^{z_{2,k}}\right)}{1 + \sum_{k=1}^{K-1} e^{z_{2,k}}} + \sum_{k=1,k\neq j}^{K} \frac{y_k e^{z_{2,j}}}{1 + \sum_{k=1}^{K-1} e^{z_{2,k}}} + \frac{y_j e^{z_{2,j}} - y_j e^{z_{2,j}}}{1 + \sum_{k=1}^{K-1} e^{z_{2,k}}} \\
&= -y_j + \sum_{k=1}^{K} \frac{y_k e^{z_{2,j}}}{1 + \sum_{k=1}^{K-1} e^{z_{2,k}}} \\
&= -y_j + \frac{e^{z_{2,j}}}{1 + \sum_{k=1}^{K-1} e^{z_{2,k}}},
\end{aligned}$$

where we used that $\sum_{k=1}^{K} y_k = 1$. That is, $\mathbf{q}_2 = \tilde{\gamma}(\mathbf{z}_2) - \mathbf{y}^{:K-1}$.

It remains to compute $\mathbf{p}_1$. We have

$$\begin{aligned}
\mathbf{p}_1 &= A_1^T \mathbf{q}_2 \\
&= \mathbb{A}_{d,K}[\mathbf{z}_1]^T (\tilde{\gamma}(\mathbf{z}_2) - \mathbf{y}^{:K-1}) \\
&= \begin{pmatrix} \mathbf{e}_1 (\mathbf{z}_1^{:1})^T & \cdots & \mathbf{e}_{K-1} (\mathbf{z}_1^{:1})^T \end{pmatrix}^T (\tilde{\gamma}(\mathbf{z}_2) - \mathbf{y}^{:K-1}) \\
&= \begin{pmatrix} \mathbf{z}_1^{:1} \mathbf{e}_1^T \\ \vdots \\ \mathbf{z}_1^{:1} \mathbf{e}_{K-1}^T \end{pmatrix} (\tilde{\gamma}(\mathbf{z}_2) - \mathbf{y}^{:K-1}) \\
&= \begin{pmatrix} (\tilde{\gamma}(\mathbf{z}_2)_1 - y_1)\, \mathbf{z}_1^{:1} \\ \vdots \\ (\tilde{\gamma}(\mathbf{z}_2)_{K-1} - y_{K-1})\, \mathbf{z}_1^{:1} \end{pmatrix}.
\end{aligned}$$

We summarize the whole procedure next.

Initialization:

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

Forward layer loop:

$$\begin{gathered}
\mathbf{z}_2 := g_1(\mathbf{w}_1; \mathbf{z}_1) = \mathcal{W}_1 \mathbf{z}_1^{;1} \\
\begin{pmatrix} A_1 & B_1 \end{pmatrix} := \boldsymbol{J}_{g_1}(\mathbf{w}_1; \mathbf{z}_1) = \begin{pmatrix} \mathbb{A}_{d,K-1}[\mathbf{z}_1] & \mathbb{B}_{d,K-1}[\mathbf{w}_1] \end{pmatrix} \\
\mathbf{z}_3 := g_2(\mathbf{z}_2) = \tilde{\gamma}(\mathbf{z}_2) \\
B_2 := \boldsymbol{J}_{g_2}(\mathbf{z}_2) = \mathbb{C}_K[\mathbf{z}_2]
\end{gathered}$$

Loss:

$$\begin{aligned}
\mathbf{z}_4 &:= \ell_{\mathbf{y}}(\mathbf{z}_3) = -\sum_{i=1}^{K} y_i \log z_{3,i} = -\sum_{i=1}^{K} y_i \log \tilde{\gamma}\left(\mathcal{W}_1 \mathbf{x}_1^{;1}\right)_i \\
\mathbf{q}_3 &:= \nabla \ell_{\mathbf{y}}(\mathbf{z}_3) = \left[-\frac{y_i}{z_{3,i}}\right]_{i=1}^{K}.
\end{aligned}$$

Backward layer loop:

$$\begin{aligned}
\mathbf{q}_2 &:= B_2^T \mathbf{q}_3 = \mathbb{C}_K[\mathbf{z}_2]^T \mathbf{q}_3 \\
\mathbf{p}_1 &:= A_1^T \mathbf{q}_2 = \mathbb{A}_{d,K-1}[\mathbf{z}_1]^T \mathbf{q}_2
\end{aligned}$$

Output:

$$\nabla f_{\mathbf{x},\mathbf{y}}(\mathbf{w}) = \mathbf{p}_1 = \begin{pmatrix} \left(\tilde{\gamma}\left(\mathcal{W}\mathbf{x}^{;1}\right)_1 - y_1\right) \mathbf{x}^{;1} \\ \vdots \\ \left(\tilde{\gamma}\left(\mathcal{W}\mathbf{x}^{;1}\right)_{K-1} - y_{K-1}\right) \mathbf{x}^{;1} \end{pmatrix}.$$

6.2.3 MNIST dataset

We will use the MNIST (https://en.wikipedia.org/wiki/MNIST_database) dataset. Quoting Wikipedia (https://en.wikipedia.org/wiki/MNIST_database):

> The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning. It was created by "re-mixing" the samples from NIST's original datasets. The creators felt that since NIST's training dataset was taken from American Census Bureau employees, while the testing dataset was taken from American high school students, it was not well-suited for machine learning experiments. Furthermore, the black and white images from NIST were normalized to fit into a 28x28 pixel bounding box and anti-aliased, which introduced grayscale levels. The MNIST database contains 60,000 training images and 10,000 testing images. Half of the training set and half of the test set were taken from NIST's training dataset, while the other half of the training set and the other half of the test set were taken from NIST's testing dataset.

Here is a sample of the images:

(Source (https://commons.wikimedia.org/wiki/File:MnistExamples.png))

We first load the data and convert it to an appropriate matrix representation. The data can be accessed with `Flux.Data.MNIST`.

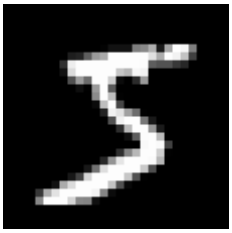
In [21]:
```
imgs = MNIST.images()
labels = MNIST.labels()
length(imgs)
```

Out[21]: 60000

For example, the first image and its label are:

In [22]:
```
imgs[1]
```

Out[22]:



In [23]:
```
labels[1]
```

Out[23]: 5

We first transform the images into vectors using `reshape`.

In [24]:
```
?reshape
```

search: **reshape** p**r**omot**e**_**shape**

Out[24]:
```
reshape(A, dims...) -> AbstractArray
reshape(A, dims) -> AbstractArray
```

Return an array with the same data as `A`, but with different dimension sizes or number of dimensions. The two arrays share the same underlying data, so that the result is mutable if and only if `A` is mutable, and setting elements of one alters the values of the other.

The new dimensions may be specified either as a list of arguments or as a shape tuple. At most one dimension may be specified with a `:`, in which case its length is computed such that its product with all the specified dimensions is equal to the length of the original array `A`. The total number of elements must not change.

Examples

```
jldoctest
julia> A = Vector(1:16)
16-element Array{Int64,1}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
 16

julia> reshape(A, (4, 4))
4×4 Array{Int64,2}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> reshape(A, 2, :)
2×8 Array{Int64,2}:
 1  3  5  7   9  11  13  15
 2  4  6  8  10  12  14  16

julia> reshape(1:6, 2, 3)
2×3 reshape(::UnitRange{Int64}, 2, 3) with eltype Int64:
 1  3  5
 2  4  6
```

In [25]:
```
reshape(Float32.(imgs[1]),:)
```

Out[25]:
```
784-element Array{Float32,1}:
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 ⋮
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
```

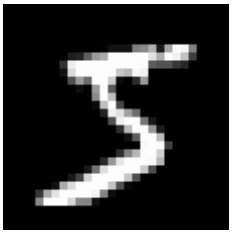
Using a list comprehension and `reduce` with `hcat`, we do this for every image in the dataset.

In [26]:
```
Xtrain = reduce(hcat, [reshape(Float32.(imgs[i]),:) for i = 1:length(imgs)]);
```

We can get back the original images by using `reshape` again.

In [27]:
```
Gray.(reshape(Xtrain[:,1],(28,28)))
```

Out[27]:



We also convert the labels into vectors. We use [one-hot encoding (https://fluxml.ai/Flux.jl/stable/data/onehot/)](), that is, we convert the label `0` to the standard basis $\mathbf{e}_1 \in \mathbb{R}^{10}$, the label `1` to $\mathbf{e}_2 \in \mathbb{R}^{10}$, and so on. The functions `onehot` and `onehotbatch` perform this transformation, while `onecold` undoes it.

In [28]:
```
?onehotbatch
```

```
search:
```

Out[28]:
```
onehotbatch(ls, labels[, unk...])
```

Return a `OneHotMatrix` where `k` th column of the matrix is `onehot(ls[k], labels)` .

If one of the input labels `ls` is not found in `labels` and `unk` is given, return `onehot(unk, labels)` (@ref) ; otherwise the function will raise an error.

Examples

```
jldoctest
julia> Flux.onehotbatch([:b, :a, :b], [:a, :b, :c])
3×3 Flux.OneHotMatrix{Array{Flux.OneHotVector,1}}:
 0  1  0
 1  0  1
 0  0  0
```

In [29]:
```
?onecold
```

```
search: SkipConnection ExponentialBackOff component_centroids
```

Out[29]:
```
onecold(y[, labels = 1:length(y)])
```

Inverse operations of `onehot` (@ref).

Examples

```
jldoctest
julia> Flux.onecold([true, false, false], [:a, :b, :c])
:a

julia> Flux.onecold([0.3, 0.2, 0.5], [:a, :b, :c])
:c
```

For example, on the first label we get:

In [30]:
```
onehot(labels[1], 0:9)
```

Out[30]:
```
10-element Flux.OneHotVector:
 0
 0
 0
 0
 0
 1
 0
 0
 0
 0
```

In [31]:
```
onecold(ans, 0:9)
```

Out[31]:
```
5
```

We do this for all labels simultaneously.

In [32]:
```
ytrain = onehotbatch(labels, 0:9);
```

We will also use a test dataset provided in MNIST to assess the accuracy of our classifiers. We perform the same transformation.

In [33]:
```
test_imgs = MNIST.images(:test)
test_labels = MNIST.labels(:test)
length(test_labels)
```

Out[33]:
```
10000
```

In [34]:
```
Xtest = reduce(hcat,
    [reshape(Float32.(test_imgs[i]),:) for i = 1:length(test_imgs)])
ytest = onehotbatch(test_labels, 0:9);
```

6.2.4 Implementation

We appeal to multinomial logistic regression (https://en.wikipedia.org/wiki/Multinomial_logistic_regression) to learn a classifier for the MNIST data. Here the model takes the form of an affine map from $\mathbb{R}^{784}$ to $\mathbb{R}^{10}$ (where $784 = 28^2$ is the size of the images in vector form and 10 is the dimension of the one-hot encoding of the labels) composed with the softmax (https://en.wikipedia.org/wiki/Softmax_function) function which returns a probability distribution over the 10 labels. The loss function is the cross-entropy (https://en.wikipedia.org/wiki/Cross_entropy#Cross-entropy_loss_function_and_logistic_regression), as we explained above.

In Flux, composition of functions can be achieved with `Chain` .

In [35]:
```
?Chain
```

```
search: Chain batched_adjoint channelview
```

Out[35]:
```
Chain(layers...)
```

Chain multiple layers / functions together, so that they are called in sequence on a given input.

`Chain` also supports indexing and slicing, e.g. `m[2]` or `m[1:end-1]`. `m[1:3](x)` will calculate the output of the first three layers.

Examples

```
jldoctest
julia> m = Chain(x -> x^2, x -> x+1);

julia> m(5) == 26
true

julia> m = Chain(Dense(10, 5), Dense(5, 2));

julia> x = rand(10);

julia> m(x) == m[2](m[1](x))
true
```

Hence our model is:

In [36]:
```
m = Chain(
    Dense(28^2, 10),
    softmax
)
```

Out[36]:
```
Chain(Dense(784, 10), softmax)
```

At initialization, the parameters are set randomly.

For example, one the first sample, we get the following probability distribution over labels:

In [37]:
```
m(Xtrain[:,1])
```

Out[37]:
```
10-element Array{Float32,1}:
 0.11419689
 0.08392158
 0.1407485
 0.032359276
 0.11348817
 0.10049408
 0.1500063
 0.053701017
 0.12868376
 0.08240051
```

We also define a function which computes the accuracy of the predictions.

In [38]:
```
accuracy(x, y) = mean(onecold(m(x), 0:9) .== onecold(y, 0:9));
```

With random initialization, the current accuracy on the test dataset is close to 10%, as one would expect from a purely random guess among 10 choices.

In [39]:
```
accuracy(Xtest, ytest)
```

Out[39]:
```
0.052
```

We are now ready to make mini-batches and set the parameters of the optimizer.

In [40]:
```
loader = DataLoader(Xtrain, ytrain; batchsize=128, shuffle=true);
```

In [41]:
```
loss(x, y) = crossentropy(m(x), y)
ps = params(m)
opt = ADAM()
evalcb = () -> @show(accuracy(Xtest,ytest));
```

We run ADAM for 10 epochs. You can check for yourself that running it much longer does not lead to much improvement.

In [42]:
```
@time train!(loss, ps, ncycle(loader, Int(1e1)), opt, cb = throttle(eval
cb, 2))
```

```
accuracy(Xtest, ytest) = 0.0608
accuracy(Xtest, ytest) = 0.9254
  8.873735 seconds (18.12 M allocations: 5.078 GiB, 4.46% gc time)
```

The final accuracy is:

In [43]:
```
accuracy(Xtest, ytest)
```

Out[43]:
```
0.9257
```

To make a prediction, we use `m(x)` which returns a probability distribution over the 10 labels. The function `onecold` then returns the label with highest probability.

In [44]:
```
m(Xtest[:,1])
```

Out[44]:
```
10-element Array{Float32,1}:
 1.4131784f-5
 2.004657f-10
 2.7680435f-5
 0.00446238
 8.006697f-7
 2.7219783f-5
 1.4060293f-9
 0.9950832
 2.7037542f-5
 0.00035753092
```

In [45]:
```
onecold(ans, 0:9)
```

Out[45]:
```
7
```

The true label in that case was:

In [46]:
```
onecold(ytest[:,1], 0:9)
```

Out[46]:
```
7
```