Deep learning for image analysis

Linus Falk

April 19, 2023

Contents

1 A linear classifier

This lecture will describe the linear classifier and ways to train it.

Recap deep learning: "end to end"

Working with deep learning can reduce the number of steps that are involved in image analysis comparing it to traditional approaches that are more human driven (engineering and prior knowledge based). The advantages with the "end to end" approach in deep learning is that we remove the prior knowledge needed for the specific case, that also removes any bias when starting to solve the problem also, since there is no pre-formed idea of how the result should look like. This is in the case were the data set can be consider non biased and non skewed.

The disadvantage of this is that it is typically very data hungry and needs large data sets in order to train well. The deep learning method is data driven and suffers if the data is not satisfying. Traditional image analysis methods c.

Problem formulation

Given a image we have an array of X $\times$ Y $\times$ 3 values representing the pixels in the image (3 colors). Images poses many challenges since the image is a 2D representation of a 3D object in a environment that can change.

- **viewpoint variations** change of camera angle
- **Illumination variations**, environment
- **Deformations**, the object may deform/
- **Occlusions**
- **Background clutter**
- **Intraclass variations**, many different looking cats...

So in contrast to sorting numbers, there is no simple solution to solve this problem with hard-coding.

```
        def predict(image):
                # no clue....
                return class_label
```

Data-driven approach

One way to solve it is to collect a lot of images and corresponding labels. Use a machine learning method to train a classifier and then evaluate the result on a test set of images.

The task is to design a classifier: f(x,**W**) that can tell us which class $y_i \in \{1, 2, ..., N\}$ an input image x_i belongs to.

1. Select a classifier type,
   - a linear affine classifier for example: y = Wx+b
2. select a performance measure, loss function
   - Hinge loss
   - Negative Log-likelihood
3. Learning: find the parameters $\mathbf{W} = \{W, b\}$ which maximizes the performance, minimizes the overall loss

Multiclass SVM loss

If given an example image with label (x_i,y_i), where x_i is the image and y_i is the label. The shorthand for the score vector: s $= f(x_i, W)$. Then the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1) \tag{1}$$

The full training loss is the mean over all examples in the training data:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i \tag{2}$$

Softmax classifier (Multinomal logistic regression)

The scores are unnormalized log probabilities of the classes: $s = f(x_i, W)$.

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \tag{3}$$

We want to maximize the log likelihood (or minimize the loss function, the negative log likelihood)

$$L_i = -\log P(Y = y_i|X = x_i) \tag{4}$$

and in summary we have:

$$L_i = -\log(\frac{e^{s_k}}{\sum_j e^{s_j}}) \tag{5}$$

Learning

How do we minimize the loss over the training data then:

- arg min loss(training data)
- Follow the slope, gradient descent

With the second alternative we want to follow the slope like a blind person finding its way down from the hills. In the 1D case this is done with the derivative, but in the multidimensional case it's done with gradients (partial derivatives). **Gradient descent** can be used to minimize the loss L by:

1. Initiliaze the weights $\mathbf{W}_0$
2. Compute the gradient with respect to W, ∇L/($\mathbf{W}_k; x) = (\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, ...)$
3. Take a small step in the negative direction of the gradient: $\mathbf{W}_{k+1} = \mathbf{W}_k -$ stepsize $\cdot \nabla L$
4. Iterate from (2) until convergence

Linear regression and classification

We will distinguish between two types of problems: regression and classification. In regression, the output y is a continuous quantity, for example a temperature, currency or distance. In classification the output is a discrete class label, for example: true/false and cat/dog/horse etc. But what is a good model? A good model is the one which **minimizes** our **Loss function**, so to answer the question we must know what a good Loss function is to begin with.

The statistical approach

One common approach is the statistical **maximum likelihood** where we make the assumtion that each data points can be described by a linear model + some noise with a normal distribution. The probability density function or PDF for the scalar Normal/Gaussian distribution:

$$\mathcal{N}(z;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(z-\mu)^2}{2\sigma^2}} \tag{6}$$

Where μ is the mean or expected value of the distribution, σ is the standard deviation and σ^2 is the variance. $z \sim \mathcal{N}(z;\mu,\sigma^2)$ means that z is a Normal/Gaussian random variable with the mean μ and variance σ^2, $\sim$: "distributed according to"

Maximum likelihood

A linear model with Gaussian noise can be modeled with:

$$y_i = wx_i + b + \epsilon_i, \quad \epsilon_i \sim \mathcal{N}(0,\sigma^2), \quad i = 1,\dots,n \tag{7}$$

We can also express this as a probability:

$$p(y_i|x_i,w,b) = \mathcal{N}(y_i; wx_i + b, \sigma^2) \tag{8}$$

By picking the weights w and bias b that makes the data as likely as possible.

$$\hat{w},\hat{b} = \arg\max\, p(y_1,\dots,y_n|x_1,\dots,x_n,w,b) \tag{9}$$

We assume here that all ϵ_i are independent. y and z are also independent so: $p(y|z) \Rightarrow p(y)p(z)$

$$p(y_1,\dots,y_n|x_1,\dots,x_n,w,b) = \prod_{i=1}^{n}(y_i - (wx_i + b))^2 \tag{10}$$

And the loss function is given by

$$L(y,\hat{y}) = \sum_{i=1}^{n}(y_i - \hat{y_1})^2 = ||y - \hat{y}||^2 \tag{11}$$

Data driven approach to image classification

The task at hand is to design a classifier f(x,**w**) that can tell us which class an image x_i belongs to. We formulate an approach:

1. Select a classifier type:
   - Start off with a linear (affine) classifier y=Wx+b
2. We then select a performance measure
   - SVM/hinge loss or Softmax + NLL loss
3. We then need for our data set the parameters W which maximizes the performance, which means minimize the overall loss. We do this by
   - Solve it with gradient descent (the learning part)

Another task is to desing a regression model f(x,W) that tells us the value of $y_i \in \mathbb{R}$ given a sample x_i

1. Select a regression model
   - We start with a linear (affine) model y = Wx +b

2. Then select a performance measure
    - Sum of squared errors for example
3. Then we want for this data set find the parameters W which maximizes the performance aka minimize the loss
    - Solve the gradient descent

The limitations of linear classifiers

Some problem are difficult to solve with linear classifiers, since they are linear. A good way to see if it's possible to use a linear classifier is to have a look in the geometric space between the input variable and the output. Is it possible to fit a straight line through the data?

Neuaral networks - stacked non-linear classifiers

By stacking multiple linear classifiers and adding non-linear activation functions between them, we can do better. Now it's possible to fit non-linear data to the mode. A stacked linear classifier can be expressed as:

$$f(x) = f_2(f_2(x)) = W_2W_1c = W^*x \tag{12}$$

This is however still linear so we need to had an activation function h(x). We then have a generalized linear classifier f(x) = h(Wx). The Logistic regression is of this type with a sigmoid activation function. A stacked non-linear classifier:

$$f(x) = f_2(h(f_1(x))) = W_2h(W_1x) \tag{13}$$

This gives us a lot more power and versatility model, known as a feed forward artificial neural network (ANN). Hera are some other activation functions:

- sigmoid(x) = $\frac{1}{1+e^{-x}}$
- tanh(x) = $\frac{e^x-e^{-x}}{e^x+e^{-x}} = 2sigmoid(2x) - 1$
- ReLu(x) = max(0,x)

2 Feed forward neural networks: Backpropagation

This lecture will cover how backpropagation works, how to compute the derivatives and some implementation.

Stepsize

Stepsize also know as the learning rate controls how large steps we take and greatly affects the training of our model. Using a to small stepsize will cause the convergence to be very slow, it can also be that we get stuck in a local minimum. Using a to big learning rate/stepsize will cause overshooting, bouncing around the minumum but never get there. Or in the worse case it can cause divergence.

Neural network - construction

An artificial neural network is a sequential construction of several generalized linear regression models. It consists of **Inputs** (1, $x_1, x_2, \ldots, x_p$), **Hidden units** (1, $q_1, q_2, \ldots, q_U$) and an output $\hat{y}$

$$
\begin{aligned}
q_1 &= h(b_1^{(1)} + \sum_{j=1}^{p} W_{1j}^{(1)} x_j) \\
q_2 &= h(b_2^{(1)} + \sum_{j=1}^{p} W_{2j}^{(1)} x_j) \\
&\vdots \qquad\qquad \vdots \\
q_U &= h(b_U^{(1)} + \sum_{j=1}^{p} W_{Uj}^{(1)} x_j)
\end{aligned}
\tag{14}
$$

$$
\hat{y} = b^{(2)} + \sum_{i=1}^{U} W_i^{(2)} q_i \tag{15}
$$

In vector representation:

$$
\begin{aligned}
W^1 = \begin{bmatrix} W_{11}^{(1)} & \dots & W_{1p}^{(1)} \\ \vdots & & \vdots \\ W_{U1}^{(1)} & \dots & W_{Up}^{(1)} \end{bmatrix}, b^{(1)} = \begin{bmatrix} b_1^{(1)} \\ \vdots \\ b_U^{(1)} \end{bmatrix}, q = \begin{bmatrix} q_1 \\ \vdots \\ q_U \end{bmatrix} \\
b^{(2)} = \begin{bmatrix} b^2 \end{bmatrix}, W^{(2)} = \begin{bmatrix} W_1^{(2)} \dots W_U^{(2)} \end{bmatrix} \\
\hat{y} = W^{(2)} q + b^2,
\end{aligned}
\tag{16}
$$

Deep neural network

A deep network with several layers will learn better but needs more training. A deep neural network with L can be written like this:

$$
\begin{aligned}
q^{(0)} &= \mathbf{x} \\
q^{\ell} = h(\mathbf{W}^{(\ell)} q^{(\ell-1)} + \mathbf{b}^{(\ell)}), &\quad \ell = 1, \dots, L-1 \\
\hat{y} &= \mathbf{W}^{(L)} q^{(L-1)} + \mathbf{b}^{(L)}
\end{aligned}
\tag{17}
$$

All the weight matrices and offset vectors in all of the layers are the parameters of the network:

$$
\theta = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(L)}, \mathbf{b}^{(L)}\} \tag{18}
$$

These parameters constitutes the parametric model $\hat{y} = f(\mathbf{x}; \theta)$ If the number L is large we call this a *deep neural network*.

Unconstrained numerical optimization

When training the neural network we are considering the following optimization problems:

$$
\hat{\theta} = \arg\min_{\theta} J(\theta), \quad J(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{x}_i, \mathbf{y}_i, \theta) \tag{19}
$$

The *global objective function* or "cost" J is the average loss over the training set. The best solution that can be found will be $\hat{\theta}$ which is the global minimizer $(\hat{\theta}^A)$. This global minimizer is often really hard to find and we therefore have to settle for some of the local minimizers $(\hat{\theta}^A, \hat{\theta}^B, \hat{\theta}^C)$.

Example: Iterative solution (gradient descent)

1. Pick a θ_0
2. While (not converged)
   - Update $\theta_{t+1} = \theta_t - \gamma \mathbf{g}_t$
   - Update t := t+1

$\gamma \in \mathbb{R}^+$ is the step length or the learning rate.

How to compute derivatives - Backpropagation

For every step during the optimization we need to calculate the gradient:

$$\mathbf{g}_t = \nabla_\theta J\theta = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta L_i(\mathbf{x}_i, \mathbf{y}_i, \theta) \tag{20}$$

But how do we compute these partial derivative? $\nabla_\theta L_j = (\frac{\partial L_j}{\partial w_1}, \frac{\partial L_j}{\partial w_2}, \ldots)$

We using the chain rules and the fact that derivatives propagating backwards up through the net: $\frac{\partial L}{\partial \text{input}} = \frac{\partial L}{\partial \text{output}} \frac{\partial \text{output}}{\partial \text{input}}$. Using a computational graph often helps simplify the understanding and work.

Compute partial derivatives

We need to calculate the gradient, which is a vector of partial derivatives of the Loss function with respect to the weights $\nabla_\theta L_j = (\frac{\partial L_j}{\partial w_1}, \frac{\partial L_j}{\partial w_2}, \ldots)$. Here the Loss is given by a rather nasty looking expression involving weight, images and outputs:

$$L(\text{Net}(x_i, \theta) = L(s(W_3 h(W_2 h(W_1)))), y_i) \tag{21}$$

Which is for a given set of images: $\{x_i\}$ and the correct output $\{y_i\}$, the variables are all our model parameters: weight and offsets. If we assuming in total m number of tunable parameters, then this is a function:

$$f : R^m \to R \tag{22}$$

The partial derivative of f at a point $\mathbf{a} = (a_1, \ldots, a_j, \ldots, a_m)$ with respect to the j-th variable x_j is the following expression:

$$\frac{\partial}{\partial x_j} f(\mathbf{a}) = \lim_{h \to 0} \frac{f(a_1, \ldots, a_j + h, \ldots, a_m) - f(a_1, \ldots, a_j, \ldots, a_m)}{h} \tag{23}$$

This tells us how the function f change, as we move a little bit, the length h, along the dimension j. There are two ways to do this:

- The **numerical gradient**: which is an approximation, slow to compute but easy to write.
- The **analytic gradient**: which is fast to compute and exact but quite error prone.

In practice we can derive the analytic gradient, check the implementation with a numerical gradient to see if it works. To derive the analytic gradients it is simply putting the chain rule to use. Since our function f is a composition of other functions f(g(h(...))) we compose the problem into smaller simpler problems:

$$\begin{aligned} f(x, y, z) &= (x + y) * z \\ f(q, z) &= q * z \\ q(x, y) &= x + y \\ q(x, y) &= x + y \\ \frac{\partial f}{\partial x} &= \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} \end{aligned} \tag{24}$$

The gradient (propagating backwards on the input side) is the local gradient of the node multiplied multiplied with the gradient arriving to the node from the output side. If the node is connected to multiple output nodes, sum up the gradients of the different paths. When using ReLu activation only pass gradients to the positive outputs. With deep neural networks there are many values multiplied together and this can lead to problems with vanishing or exploding gradients. A ReLu that never turns on during the training, will never move (zero gradient) these are called "a dead ReLu".

Summary

- **Neural network (NN)**: A nonlinear *parametric* model constructed by stacking several linear models with intermediate nonlinear activation functions.
- **Activation function**: A nonlinear scalar function applied to each output element of the linear models in a NN.
- **Gradient descent**: An iterative optimization algorithm where we at each iteration take a step proportional to the negative gradient.
- **Learning rate**: A scalar and tunable parameter which decide the length of each gradient step in gradient descent.
- **Back-propagation**: An efficient method that is based on the chain rule to compute the gradients.

3 Stochastic Gradient Descent

The deep learning optimization problem are given by:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} L(f(\mathbf{x}_i, \theta)y_i) = \frac{1}{n}\sum_{i=1}^{n} L_i(\theta) \tag{25}$$

This problem introduces some difficulties since it is non-convex, of large scale (large n and many dimensions of θ) and the data i often, if not always: noisy. For this there are two main classes of optimization methods:

- deterministic, looking at the whole bath of training data
- Stochastic, looking at a random subset of the training data each time, (c.f. bagging random forests)

Sometimes the deterministic model might terminate at the wrong solution when it finds a local minimum or saddle points. Another disadvantage is the computational cost of computing the gradients over the entire dataset. By instead using stochastic gradient descent we introduce some randomness by using a randomly picked subset of the entire dataset. This approach can help avoiding getting stuck in local minimas and therefor finding better solutions. The disadvantages might be that we pcik the minibatches in order an therefor don't have a representative batch for the whole dataset. It is therefor important to pick data points at random (without replacement). One way to implement stochastic gradient descent is to randomly reshuffle the data before we dived it into minibatches. Then after each epoch we do another reshuffling and a another pass through the dataset.

Minibatch Stochastic gradient descent (SGD)

```
1. Initialize $\theta_0$, set $k \leftarrow 1$, choose a batch size $n_b$ and number of Epochs $E$
2. for j = 1 to $\frac{n}{n_b}$
     • Approximate the dradient of the loss function using the current minibatch
```

4 Convolutional neural networks