Neurális hálózatok házi feladat beszámoló

Pintér Bálint (I6QS0K), Szilágyi Gábor (NOMK01)

Koncentrált paraméterű RF szűrő optimalizációja aktív tanulással

1. Bevezetés

1.1. Aktív tanulás

A legtöbb neurális hálózatokat használó megoldás olyan problémára irányul, ahol sok rendelkezésre álló adat alapján kell a hálót betanítani egy feladat elvégzésére. Az a legtöbb esetben teljesül, hogy még több tanító adat felhasználásával jobb hálót lehetne tanítani, de ennek az extrém esetére tud megoldást nyújtani az aktív tanulás. A nem adathiányos problémáknál a rendelkezésre álló, címkézett adatpontok nagy részét felhasználva szokás tanítani a hálót, majd a fennmaradó adatpontokon ellenőrizni a háló teljesítőképességét olyan esetekre, amikkel nem találkozott a tanulás során. Az aktív tanulás folyamata ettől merőben eltér.

Aktív tanulás kiindulási helyzete, hogy nagyon sok címkézetlen adat áll rendelkezésre, de az egyes adatpontok címkézése rendkívül költséges. A címkézés költsége miatt végeredményben az a cél, hogy azt minél kevesebbszer kelljen elvégezni a tanulás során. A tanulási folyamat közben az eddig megkapott kevés címkézett adatpont alapján a háló jelöl ki következőnek címkézésre azt, amelyik várhatóan a leghasznosabb lesz számára. A hasznosság becslésére több megközelítés is létezik, erre a későbbiekben visszatérünk.

Az aktív tanulás egyik alesete a Bayes-optimalizáció. Itt nem egy osztályozót tanítunk minél kevesebb címkézett adat alapján, hanem egy „fekete doboz” függvény maximumát keressük a függvény minél kevesebb kiértékelése mellett. Ez a különbség már befolyásolni fogja a következőnek megcímkézendő adat választását, ami ebben az esetben a következő paraméterértékek megválasztását jelenti, ahol kiértékeljük a függvényt.

1.2. Az optimalizálandó probléma

Az optimalizálandó probléma egy koncentrált paraméterű rádiófrekvenciás (RF) szűrő elemeinek megadása, azaz méretezése. Egy koncentrált paraméterű elemekből felépülő szűrő alapesetben ellenállásból, tekercsekből és kondenzátorokból áll, de léteznek aktív elemet, azaz erősítőt is tartalmazó szűrők – de ez már kívül esik a vizsgálatunkon, csak passzív eszközökkel foglalkozunk. Feltesszük továbbá, hogy a vizsgálatunk során csillapítással nem foglalkozunk, így ellenállást sem használunk a szűrőnkhöz.

Szűrőtervezéskor mindig egy előre megadott specifikációból indulunk ki, ebben elő van írva azon frekvenciatartományok csoportja, ahol a szűrni kell, illetve ahol csillapítás nélkül kell az RF teljesítményt áteresztení. Ennek megfelelően beszélhetünk záró és áteresztő sávokról, ahol a szűrő impedanciakarakterisztikájának rendre nagynak, illetve kicsinek kell lennie, ehhez a tekercsek és kondenzátorok soros és párhuzamos rezonanciáit használjuk ki. (S paraméter majd később).

A valóságban mind a tekercsek, mind a kondenzátorok 3D-s objektumok, induktivitásuk és kapacitásuk anyagparamétertől és geometriai méretektől függenek. Anyagparamétertől függés alatt a kondenzátorok dielektrikumában a relatív permittivitását, illetve a tekercsek belsejében a közeg relatív permeabilitását kell érteni – bár utóbbi nem jellemző RF szűrők esetében. Az anyagparaméter tehát egy fix érték, amit nem egyszerű változtatni, méretezés céljából ez nem járható út. Megoldást a geometriai méretek optimális megválasztása jelent, ami utat nyit az alapesetektől, az analitikus formulákkal egyszerűen kiszámolható geometriáktól való eltéréshez. Bár ezekben az alapesetekben, a síkkondenzátorban és az egyenes tekercsben, a szolenoidban is megjelennek geometriai jellemzők, a

$$C = \varepsilon_0 \varepsilon_r \frac{A}{d}$$
$$L = \mu_0 \mu_r \frac{N^2 A}{l}$$

formulák egyrészt csak közelítések, azaz nem írják le pontosan az L és a C értékeket, másrészt sokszor nem kivitelezhető az általuk leírt komponens.

A feladat innentől az, hogy olyan geometriai méreteket találjunk, amivel a komponensek pont az általunk kívánt L és C értékeket mutassák, ezzel a specifikációnak eleget tudjunk tenni. Esetünkben a nem analitikusan kiszámolható értékekhez numerikus megoldást kell keresni, amire egy jó megoldás, ha 3D véges elemes szimulációt készítünk. Egyegy szimuláció futási ideje a megkövetelt precizitás és a rendelkezésünkre álló erőforrás függvényében változhat, de mindenképp több mint csupán egyetlen egyenlet kiértékelése. Éppen ezért szükséges az optimalizálást úgy végezni, hogy szempont legyen a minél kevesebbszer történű kiértékelés.

A fent leírtak az RF szűrőtervezés témakörében egy releváns problémát takarnak, munkánkban azonban nem 3D véges elemes szimulációkat használunk, nem geometriai paraméterek optimalizálását végezzük el. Tesszük mindezt egyrészt azért, mert a szimulációs szoftverek programozott vezérlése **kibaszott nehéz**, másrészt a neurális hálók szempontjából teljesen lényegtelen, hogy a kiértékelt függvényben mi történik. Azonban maga az optimalizálás amit csinálunk alkalmas egy ilyen feladatra, és érdemben tudja a tervezőt segíteni munkája során. Visszatérni tehát a koncentrált paraméterű, L és C elemeket tartalmazó hálózatra az optimalizáció szempontjából nem jelent kevesebb munkát, ebben az értelemben nem jár a feladat egyszerűsítésével.

A feladathoz már csak egy lépés hiányzik, amivel a problémát át tudjuk alakítani egy szélsőérték keresésre. Ennek az alapgondolata az, hogy az elvárt és a kapott kimenet különbségét vesszük figyelembe egy büntetőfüggvény segítségével. Magát az optimalizálást ezen keresztül tudjuk megtenni, ennek a függvénynek a kifejtésére a **??** részben kerül sor.

1.3. A felhasznált könyvtár

Bayes-optimalizáció megvalósításához már léteznek elkészített könyvtárak, így azt külön nem készítettük el, hanem felhasználtuk. Az általunk használt ingyenesen elérhető és letölthető *Bayesian Optimization* könyvtár más, neurális hálózatokra készített, ugyancsak szabadon hozzáférhető könyvtárakon alapszik, itt most ennek a bemutatását adjuk.

A *Bayesian Optimization* használata rendkívül egyszerű, összesen két dolgot kell megtennünk. Elsőként példányosítjuk a *BayesianOptimization* osztályt, ami bemeneti paraméterként megkapja a maximalizálandó költségfüggvényt, illetve a határait annak a

tartománynak, ahol a maximumot keresnie kell. Fontos kiemelni, hogy esetünkben a kapott és az elvárt kimenetek közötti minimalizálás a feladat, így a költségfüggvényben át kell térni maximumhely keresésre.

A szélsőértékhely keresést ezután a *maximize( )* tagfüggvény hajtja végre. Ez a folyamat kezdetben random helyeken kiértékel pontokból indul el, majd iteratívan halad a modelltérben az optimum felé. Ennek megfelelően a *maximize( )* bemeneti paraméterei az inicializáló, véletlenszerű helyek száma és az iteráció száma. Ebben elsőként példányosul egy *UtilityFunction* nevű osztály, utána elindul az iteráció. Minden kör az előző lefutás utáni paraméterfrissítéssel, az *update_param( )*-mal kezdődik, ami a *UtilityFunction* egy tagfüggvénye. A függvénynek több felderítési típusa lehet, ezek a UCB (Upper Confidence Bounds method), az EI (Expected Improvement method) és a POI (Probability Of Improvement), alapesetben az UCB-t használjuk. Az *update_param( )*-nak három hiperparamétere van: χ, κ és κ_{decay}, az utóbbi a κ csökkentését végzi minden iterációban, de csak egy másik paraméter, κ_{delay} iterációtól kezdve. Frissítve a *UtilityFunction*-t, egy javaslattevő tagfüggvénynek, a *suggest( )*-nek adjuk át inputként, ami javasol egy új pontot, az *x_probe*-ot a következő kiértékelésre. Ebben a *suggest( )*-ben egy *argmax* vizsgálat történik egy akvizíciós függvényre, az *acq_max*-ra. Ez a vizsgálat két lépésből áll, elsőként egyenletes eloszlással felvesztünk mintavételi pontokat a *UtilityFunction*-ból a paramétertartomány határain belül, alapbeállításként 10^5 darabot. Második lépésként *L-BFGS-B* optimalizációs metódus futtatunk le minden mintavételezett pontra. Egy ciklusban kiválasztjuk a legnagyobbat, ezzel térünk vissza, ez lesz *x_probe*, a javasolt új kiértékelési hely. A *maximize( )* iterációs ciklusának harmadik lépése maga a kiértékelés, erre a *probe( )* tagfüggvény szolgál, ami az általunk megadott költségfüggvényt futtatja le a kiszámolt új *x_probe* helyen. Ezt a három iterációs lépést folytatja a kód a meghatározott iterációszámig.

Az *L-BFGS-B* optimalizáló solver nem ebben a könyvtárban van megírva, hanem a *scipy.optimizer* tartalmazza. Ez egy minimalizálási feladatokra készített másodrendű optimalizációs algoritmus, az *L-BFGS* kiterjesztése korlátok kezelésére. Feloldva a rövidítést a Limited-memory Broyden-Fletcher-Goldfarb-Shanno algoritmus egy kvázi-Newton módszer, másodrendű deriváltak közelítésére hasznos, ahol az közvetlenül nehéz kiszámolni. Konkretizálva ez azt jelenti, hogy nem számolja ki a Hesse-mátrixot, csak annak inverzét közelíti. A módszer nevében az L előtag a limitált memóriára utal, azaz mindig csak az elmúlt m lépés koordináta- és gradiensvektorát tárolja el. Ez a memóriagazdálkodás fontos, különösen az *acq_max* függvényben, ahol mind a 10^5 mintavételi pontra elvégezzük az *L-BFGS-B*-t.

A. Az általunk írt specifikáció osztály

```
import numpy as np
import math
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
from matplotlib.backends.backend_pdf import PdfPages

pi = math.pi
inf = math.inf
nan = math.nan

#ipmedance of the series LC subcircuit
def Impedance(f, lnC, lnL):
    match [math.isnan(lnC), math.isnan(lnL)]:
        case [True, True]:
            return 0.0
        case [True, False]:
            L = math.exp(lnL)
            return f*1j*2*pi*L
        case [False, True]:
            C = math.exp(lnC)
            return -1j*1/(f*2*pi*C)
        case _:
            C = math.exp(lnC)
            L = math.exp(lnL)
            ZC = -1j*1/(f*2*pi*C)
            ZL = 0+f*1j*2*pi*L
            return ZL*ZC/(ZL+ZC)

def Admittance(f, lnC, lnL):
    match [math.isnan(lnC), math.isnan(lnL)]:
        case [True, True]:
            return 0.0
        case [True, False]:
            L = math.exp(lnL)
            return 1/(f*1j*2*pi*L)
        case [False, True]:
            C = math.exp(lnC)
            return f*2*pi*C*1j
        case _:
            C = math.exp(lnC)
            L = math.exp(lnL)
            ZC = -1j*1/(f*2*pi*C)
            ZL = 0+f*1j*2*pi*L
            return (ZL+ZC)/ZL*ZC

class Spec:
    def __init__(self, starts, ends, limits, directions, margin, n):
        # starts of pass- or stopbands in Hz
        self.starts = starts
        # ends of start- or stopbands in Hz
        self.ends = ends
        # pass- or stopband threshold values
        self.limits = limits
        # pass- or stopbands? possible values: "pass" or "stop" strings
        self.directions = directions
        # number of frequency points
        self.n = n
        # margin to still punish solution that only barely satisfies the specification
        # on stopbands, with limit l and margin m, the margin range is from l*(1-m) to l
        # on passbands, with limit l and margin m, the margin range is from l to l*(1+m)
        self.margin = margin

    #def cost(self, types, values):
    def cost(self, plot=False, par1C=nan, par1L=nan, ser1C=nan, ser1L=nan, par2C=nan, par2L=
    nan, ser2C=nan, ser2L=nan, par3C=nan, par3L=nan, ser3C=nan, ser3L=nan):
        """Parameters define a ladder structure,
        where each series or parallel element consists of
        two discrete, ideal L or C components in parallel
        with each other. Default values define a perfect
        all-pass filter. In the function parameters, "parXY" means
```

```
        the value of the Y-th sub-component of the X-th parallel
        element. Similarly, "serXY" means the value of the Y-th
        sub-component of the X-th series element. The ladder starts
        with a parallel element."""
        values = [par1C, par1L, ser1C, ser1L, par2C, par2L, ser2C, ser2L, par3C, par3L, ser3C,
     ser3L]
        nval = len(values)
        nladder = int(nval/4)
        minf = (min(self.starts))
        maxf = (max(self.ends))
        # frequency axis sample points (log spacing)
        faxis = []
        # no. of frequency sample points
        #n = self.n
        factor = (maxf/minf)**((1/self.n))
        for i in range(self.n+1):
            faxis.append(minf*factor**(i))
        # reference impedance in Ohm, on both ports
        Z0 = 50
        Y0 = 1/Z0
        # array of overall S21 values at the frequency sample points
        S21 = []
        #process 1 parallel and 1 series element:
        for f in faxis:
            # ABCD parameter matrix of the whole system
            ABCD = np.matrix([[1,0],[0,1]])
            for l in range(nladder):
                # admittance of the two parallel components together from the current step of
    the ladder
                Y = Admittance(f, values[4*l+0], values[4*l+1])
                mPar = np.matrix([[1, 0],[Y, 1]])
                Z = Impedance(f, values[4*l+2], values[4*l+3])
                mSer = np.matrix([[1, Z],[0, 1]])
                ABCD = ABCD*mPar*mSer
            # 2/(A+B/Z0+C*Z0+D)
            A = ABCD.item(0,0)
            B = ABCD.item(0,1)
            C = ABCD.item(1,0)
            D = ABCD.item(1,1)
            S21.append(abs(2/(A+B/Z0+C*Z0+D)))
        # natural log of margin+1
        lnmargin=math.log(self.margin+1)
        if(plot):
            # green: passband; red: stopband; orange: ok, but close to not ok, still punished
            fig, ax = plt.subplots()
            minS21 = min(S21)
            for i in range(len(self.starts)):
                if(self.directions[i] == "pass"):
                    # region forbidden by the original specification
                    ax.add_patch(Rectangle((self.starts[i], minS21),
                                           self.ends[i]-self.starts[i],
                                           self.limits[i]-minS21,
                                           facecolor='#00aa00'))
                    # region close to original limit, but satisfying it, additional penalty
    region
                    ax.add_patch(Rectangle((self.starts[i], self.limits[i]),
                                           (self.ends[i]-self.starts[i]),
                                           self.limits[i]*self.margin,
                                           facecolor='orange'))
                else: # "stop"
                    # region forbidden by the original specification
                    ax.add_patch(Rectangle((self.starts[i], self.limits[i]),
                                           self.ends[i]-self.starts[i],
                                           1.0-self.limits[i],
                                           facecolor='#aa0000'))
                    # region close to original limit, but satisfying it, additional penalty
    region
                    ax.add_patch(Rectangle((self.starts[i], self.limits[i]*(1/(1+self.margin))
    ),
                                           (self.ends[i]-self.starts[i]),
                                           self.limits[i]*(1-1/(1+self.margin)),
                                           facecolor='orange'))
            ax.loglog(faxis, S21, 'k-')
```

```
            plt.ylabel(r'$|S_{21}|$')
            plt.xlabel(r'$f$')
            plt.savefig("plot.pdf", dpi=120, format='pdf', bbox_inches='tight')
            plt.show()
        cost = 0
        # number of freq points in the regions where the S21 is specified
        ncost = 0
        for i in range(len(self.starts)):
            for j in range(len(faxis)):
                if faxis[j]>self.starts[i] and faxis[j]<self.ends[i]:
                    ncost = ncost + 1
                    lnlimit = math.log(self.limits[i])
                    lnS21 = math.log(S21[j])
                    if self.directions[i] == "pass":
                        cost += max(0, min(-lnS21, lnlimit+lnmargin-lnS21))
                    else: # "stop"
                        cost += max(0, lnS21-lnlimit+lnmargin)
    # negative of average cost, for function maximizing
        return -cost/ncost
```