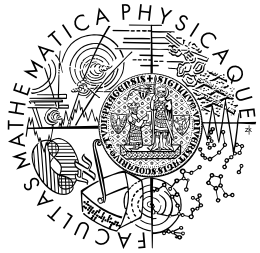


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Filip Ježek

Nástroj pro průzkum a vizualizaci modelu pro simulátor mozku Mozaik

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: Mgr. Ján Antolík, Ph.D.

Studijní program: Informatika

Studijní obor: Databáze a web

Praha 2023

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Rád bych poděkoval Mgr. Jánu Antolíkovi, Ph.D. za vedení této práce, za jeho čas a hodnotné rady. Dále bych rád poděkoval Rémy Cagnolovi a Tiboru Rózsovi z CSNG za výborné připomínky a podporu při vývoji. Rád bych také poděkoval své rodině a kamarádům.

Název práce: Nástroj pro průzkum a vizualizaci modelu pro simulátor mozku Mozaik

Autor: Filip Ježek

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: Mgr. Ján Antolík, Ph.D., Katedra softwaru a výuky informatiky

Abstrakt: Simulace biologických neuronových sítí jsou důležitým nástrojem pro porozumění tomu, jak mozek zpracovává informace. Mozaik je workflow framework, který umožňuje takové simulace vytvářet, spouštět a analyzovat. V současné době ale neexistuje snadný způsob, jak v něm vizualizovat ani síť, ani datové struktury vytvořené jednotlivými analýzami. Tato práce je webová aplikace, která umožňuje vizualizovat sítě a datové struktury uložené v datastorech pro jednotlivé simulace. Dané vizualizace lze vyhledávat pomocí komplexních dotazů, interaktivně je zkoumat a vzájemně mezi sebou srovnávat. To výrazně zefektivňuje práci výzkumníků, kteří s frameworkem Mozaik zacházejí.

Klíčová slova: vizualizace webová aplikace vědecký software biologické neuronové sítě neurověda

Title: A model inspection/visualization tool for brain simulation framework

Author: Filip Ježek

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Ján Antolík, Ph.D., Department of Software and Computer Science Education

Abstract: Simulations of biological neural networks are an important tool for understanding how the brain processes information. Mozaik is a workflow framework that allows such simulations to be created, run and analyzed. Currently, however, there is no easy way to visualize in it either the network or the data structures created by individual analyses. This work is a web application that allows users to visualize networks and data structures stored in datastores for individual simulations. The given visualizations can be searched using complex queries, interactively examined and compared with each other. This makes the work of researchers working with the Mozaik framework much more efficient.

Keywords: visualization web application scientific software biological neural networks neuroscience

Obsah

Úvod

Porozumění lidskému mozku je téma, které v posledním desetiletí nabylo ještě většího významu. Mluví se o něm jak v souvislosti s umělou inteligencí, tak s neuroprotetikami. Klíčovým přístupem k výzkumu se stává výpočetní neurověda — simulace jednotlivých vrstev neuronů, které napodobují mozkovou strukturu a funkcionalitu. CSNG (Computational Systems Neuroscience Group)[1] z Matematicko-fyzikální fakulty za tímto účelem aktivně vyvíjí nástroj Mozaik[2]. Mozaik umožňuje specifikovat, spustit a analyzovat simulace neuronových sítí, přičemž veškerá data z takové simulace se ukládají do *datastore*. Během simulace typicky běží několik algoritmů za různých podmínek, každý algoritmus pak generuje jednu nebo více *datových struktur*. Tyto datové struktury je možné načítat pomocí Pythonu a zkoumat např. v Jupyter noteboocích. Dosud ale není možnost uživatelsky přívětivě vizualizovat strukturu neuronových vrstev a spojení, ani uceleně zobrazit a filtrovat všechny datové struktury.

Cíle

Cílem této práce je návrh a implementace webové aplikace pro vizualizaci modelu a datových struktur vygenerovaných za běhu Mozaiku. Aplikace by měla poskytnout rozhraní pro dotazy nad metadaty datových struktur, každý druh datové struktury by měl mít vlastní typ vizualizace a vizualizace by měly být interaktivní. Velká část práce s výsledky analýz je komparační, v aplikaci by tedy mělo být snadné porovnávat datové struktury vytvořené s různými parametry.

Struktura práce

Nejprve si v kapitole 1 přiblížíme Mozaik. Podíváme se na základy práce s ním, ale hlavně se seznámíme se strukturou datastore, ze kterého budeme získávat data. Také je důležité popsat návaznost na předchozí práce v tomto směru. V kapitole 2 formulujeme přesné požadavky na aplikaci. Kapitola 3 bude věnována popisu architektury, jejíž implementaci pak rozvedeme v kapitole 4. Speciálně se

[1] `https://csng.mff.cuni.cz/`

[2] `https://github.com/CSNG-MFF/mozaik`

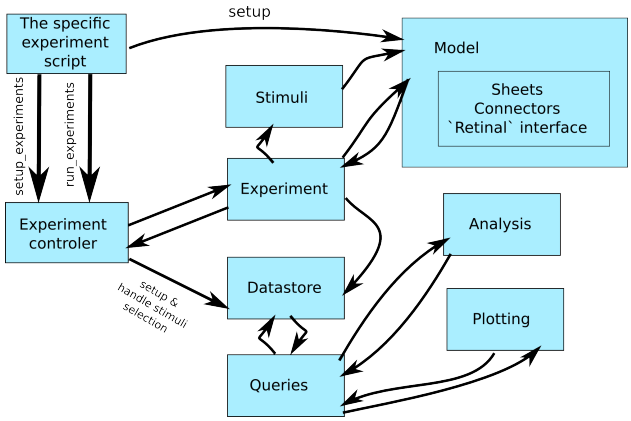
podíváme na klíčové třídy frontendu v kapitole 5. Posléze si v kapitole 6 přiblížíme zacházení s programem. Konečně v příloze B je podrobně popsáno serverové API.

Kapitola 1

Mozaik

1.1 Práce s frameworkem

Mozaik je framework pro Python, pomocí kterého lze snadno specifikovat 2-rozměrné vrstvy neuronů, nad kterými pak běží experimenty. Experimenty produkují data, která lze analyzovat různými algoritmy a zaznamenávat jejich výstupy. V Mozaik tutorialu[1] je ukázán příklad, na který se zde (v trochu zkrácené podobě) podíváme.



Obrázek 1.1 Control flow mezi jednotlivými komponentami Mozaiku[2]

Kód je rozdělen na 6 hlavních částí. Část 1 a 2 jsou definice neuronových vrstev — třída `Model` a její parametry. Stejný neuron může být součástí více vrstev na různých souřadnicích.

```
class VogelsAbbott( Model ):
  # parametry jsou nacteny z oddeleneho souboru
  required_parameters = ParameterSet({
    # definujeme 2 neuronove vrstvy
    'sheets': ParameterSet({
        'exc_layer': ParameterSet,
        'inh_layer': ParameterSet,
```

```
    })
  })

  def __init__(self, sim, num_threads, parameters):
    Model.__init__(self, sim, num_threads, parameters)
    # nacteni komponent vrstev
    ExcLayer = load_component(
      self.parameters.sheets.exc_layer.component
    )
    ...

    exc = ExcLayer(
      self,
      self.parameters.sheets.exc_layer.params
    )
    ...

    # spojeni mezi vrstvami
    UniformProbabilisticArborization(
      self,'ExcExcConnection',exc,exc,
      self.parameters.sheets.exc_layer.ExcExcConnection
    ).connect()
    ...
```

Dále je potřeba specifikovat samotné experimenty, které chceme na mozku provést.

```
def create_experiments(model):
  return [
    #Lets kick the network up into activation
    PoissonNetworkKick(model,ParameterSet({
      'duration': 8*7,
      'drive_period': 8*7.0,
      'sheet_list': ["Exc_Layer","Inh_Layer"],
      'stimulation_configuration' : {
        'component':
          'mozaik.sheets.population_selector'
          '.RCRandomPercentage',
        'params': {'percentage' : 20.0}
      },
      'lambda_list': [100.0,100.0],
      'weight_list': [0.1,0.1]
    })),
    #Spontaneous Activity
    NoStimulation(model,ParameterSet({'duration': 135.0*2}))
  ]
```

Experimenty produkují surová data, *segmenty*. Ta nám sama o sobě moc neřeknou, ale můžeme na nich spouštět analytické algoritmy. Ještě předtím ale spustíme samotnou simulaci.

```
data_store,model = run_workflow(
  'example simulation',
```

```
  VogelsAbbott,
  create_experiments
)
```

Segmenty jsou uložené v datastore a nám už nic nebrání je analyzovat.

```
def perform_analysis_and_visualization(data_store):
  analog_ids = sorted(
    param_filter_query(data_store,sheet_name="Exc_Layer") \
    .get_segments()[0].get_stored_esyn_ids()
  )
  ...


  PopulationMeanAndVar(
    data_store,ParameterSet({'ignore_nan_and_inf': False})
  ).analyse()
  ...
```

V tuto chvíli můžeme použít modul `mozaik.visualization` a vygenerovat množství rozmanitých grafů, které se uloží jako png soubory. My ale chceme data prohledat interaktivně, takže tuto funkčnost používat nebudeme a podíváme se raději na různé typy datových struktur, které se v datastore mohou nacházet.

1.2 Datastore

Datastore se na disk serializuje ve formátu pickle, což je binární protokol pro serializaci python objektů. Kromě *ADS* (analysis data structure) se v něm nachází metadata o neuronech, jejich vrstvách a stimulech. ADS lze vyhledávat pomocí jejich parametrů. ADS může být (ale nemusí) spojena s konkrétní vrstvou, stimulem, nebo konkrétním neuronem. Dělí se na několik typů, podle typu dat, které uchovávají.

1.2.1 SingleValue

Wrapper pro jedinou hodnotu, obohacenou o důležitá metadata.

1.2.2 SingleValueList

Jednorozměrný seznam hodnot.

1.2.3 PerNeuronValue

Podobná datová struktura jako SingleValueList, ale každá hodnota v seznamu odpovídá konkrétnímu neuronu.

1.2.4 PerNeuronPairValue

Seznam neuronů a matice hodnot pro každý jejich pár.

1.2.5 AnalogSignal

Jde o wrapper nad Neo `AnalogSignal` [1], jenž k němu doplňuje klíčová metadata. Neo `AnalogSignal` je dvourozměrné pole periodicky měřených hodnot. Jedna dimenze odpovídá času a druhá jednotlivým zaznamenávaným kanálům. V praxi se ale v Mozaiku do AnalogSignalu zaznamenává pouze jeden kanál, takže se hodí o něm uvažovat jako o jednorozměrném poli.

1.2.6 AnalogSignalList

Kombinace PerNeuronValue a AnalogSignal. Seznam AnalogSignalů, kde každý AnalogSignal náleží specifickému neuronu.

1.2.7 PerNeuronPairAnalogSignalList

Tentokrát jde o kombinaci AnalogSignalu a PerNeuronPairValue. Seznam neuronů a matice AnalogSignalů pro každý jejich pár.

1.2.8 ConductanceSignalList

Jedná se o rozšířený AnalogSignalList s tím, že pro každý neuron jsou zde uloženy 2 AnalogSignaly.

1.2.9 Connections

Tato datová struktura popisuje spojení mezi neurony, což jsou orientované hrany v grafu. Každá hrana má navíc ještě specifikovanou váhu a zpoždění.

1.3 Předchozí práce

Tato práce navazuje na ročníkový projekt Kateřiny Čížkové [3]. Její program byla webová aplikace, která umožňovala interaktivně zobrazit neuronové vrstvy v zadaném datastore. Původní záměr byl tento projekt rozšířit, ale, jak je v dokumentaci projektu popsáno, kolegyně zvolila ne úplně vhodnou technologii a doporučuje začít potenciálně zcela od začátku. Tato práce tedy namísto Python frameworku Bokeh používá Flask a renderování vizualizací je specifikováno až ve frontendu (více podrobností je v kapitole 4). Podobnost kódu je naprosto minimální, nicméně zejména ze začátku sloužil jako výborná inspirace.

[1] `https://neo.readthedocs.io/en/stable/api_reference.html#neo.core.AnalogSignal`

Kapitola 2

Analýza požadavků

Požadavky na aplikaci byly zadány dobře a přesně, díky tomu, že vzešly přímo od budoucích uživatelů. Aplikace byla nasazena už během vývoje a průběžně používána, což napomohlo najít nedostatky včas a uvést ji do stávající podoby.

Přenos velkého objemu dat Aplikace musí zvládnout načíst a zpracovat datastory co největší velikosti. Velikost datastoru se pohybuje v řádu gigabytů, neuronů mohou být desítky tisíc a hran desítky milionů. Je potřeba minimalizovat velikost dat pro přenos po síti, a je důležité dát si pozor na nápor na paměť serveru. Výše zmíněná instance měla k dispozici zhruba 8GB paměti — na tento limit se během provozu alespoň ze začátku naráželo často. O něco méně je to problém u klienta. Pochopitelně se zdroji nechceme plýtvat, ale stroje, na kterých se aplikace ve výsledku spouští, patří výzkumníkům a jsou výkonné.

Specifikace složek s datastore Datastore k prohlížení se nachází na straně serveru. Aplikaci lze spustit na lokálním počítači, nebo například na serveru se sdílenými adresáři více uživatelů. V obou případech se hodí zadat serveru cestu, odkud může začít procházat adresářový strom. Jednak aby se zrychlilo vyhledávání uživatele, jednak kvůli bezpečnosti v případě veřejně přístupného serveru. Aplikace nesmí akceptovat cestu k datastore, která by nebyla potomkem nakonfigurovaného kořene. Zároveň by mělo být možné specifikovat více kořenů naráz.

Přehledný výpis ADS Aplikace je potřeba kvůli zrychlení práce uživatelů. ADS by měly být zobrazeny tak, aby se šlo co nejrychleji zorientovat a najít to, co uživatel chce. ADS mají mnoho stejných parametrů, ty je potřeba umět vyfiltrovat a zobrazit jen ty, ve kterých se liší.

Filtrování a vyhledávání ADS Na předchozí požadavek navazuje nutnost mít kvalitní vyhledávací systém. ADS by mělo být možné snadno a intuitivně filtrovat a řadit. Zároveň musí být filtrovací systém dostatečně silný na to,

aby zvládal i hodně komplikované dotazy. Filtry by měly být přizpůsobené jednotlivým datovým typům parametrů.

Workspace s ADS Načítání konkrétní ADS ze serveru může zabrat netriviální množství času. Je nutné mít možnost některé ADS označit a udržovat je v paměti, aby mezi nimi bylo možné rychle přepínat. Zároveň je nutné mít možnost konkrétní ADS z paměti odstranit a uvolnit zdroje.

Vícenásobné zobrazení V určitých případech je rušivé a zdržující muset pořád přepínat mezi několika ADS, obzvlášť pokud je uživatel srovnává a hledá rozdíly a podobnosti. Tehdy je potřeba umět zobrazit více ADS naráz vedle sebe.

Sdílené nastavení Každý druh vizualizace má svá specifická nastavení. Někdy se hodí moci upravovat nastavení pro každou ADS zvlášť, ale v případě vícenásobného zobrazení musí existovat možnost upravovat nastavení pro všechny zobrazené vizualizace stejného typu naráz.

Connections Aplikace musí umět zobrazit spojení mezi neurony ve stejné vrstvě. Vrstva by měla být vizualizována jako interaktivní scatterplot.

PerNeuronValue Aplikace musí umět vizualizovat PerNeuronValue ADS. Vizualizace by měla mít stejnou podobu jako vizualizace Connections, ale neurony by měly mít přiřazenou barvu na základě své hodnoty.

PerNeuronPairValue Aplikace musí umět vizualizovat PerNeuronValue ADS. Vizualizace by měla být maticový graf.

AnalogSignalList Aplikace musí umět vizualizovat AnalogSignalList ADS.

Udržovatelnost Jeden z nejdůležitějších požadavků je udržovatelnost aplikace. Aplikace musí být napsána srozumitelně a rozšiřitelně. Musí být použity moderní technologie, které pravděpodobně brzy nezaniknou. Aplikace musí být dokumentována a důkladně otestována automatickými testy.

Kapitola 3

Návrh architektury

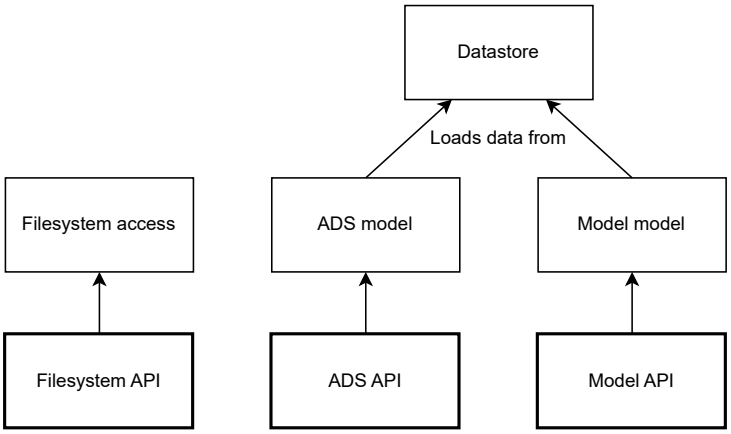
Aplikace se skládá ze dvou hlavních celků, webového klienta a serveru. Většina logiky je na straně klienta, server slouží pouze pro načítání dat z datastore.

3.1 Server

Server je minimální, chová se jako prostředník mezi datastore a klientem. Jeho součástí je modul pro souborový systém, pomocí něhož je možné procházet adresáře a vypisovat přítomné datastory. Model API načítá neurony v jednotlivých vrstvách a jejich spojení. ADS API pak načítá seznam a detaily datových struktur. Jak modelová část serveru, tak ADS část serveru kromě veřejného rozhraní zahrnují jen minimální transformaci dat do formátu vhodného pro frontend.

3.1.1 Popis API

Původní záměr byl vybudovat klasické JSON API. Během vývoje se ale ukázalo, že nejde o proveditelný nápad. Model i ADS zprávy mohou narůst do ohromné



Obrázek 3.1 Architektura serveru

velikosti (například spojení mezi dvěma vrstvami mohou být desítky milionů). Když se server pokoušel zakódovat do JSON pole takových rozměrů, narazil na limit přidělené paměti. Jako JSON se tedy posílají jen relativně krátké zprávy, většinou metadata. Hlavní objem dat se streamuje ve formátu CSV. To má benefit i v lepším využití času na straně frontendu — data se mohou začít zpracovávat už během stahování.

Speciální případ je načítání detailu ADS. Různé druhy ADS mají různě strukturovaná data s velkým objemem. Liší se jak dimenzionalita polí, tak jejich počet. API je vytvořené tak, aby frontend tyto informace nemusel předem znát. Detail API, který frontend dostane, místo některých properties může obsahovat `@link` objekty. Ty frontendu sdělí, na jaké URI nalezne daná streamovaná data a jakou mají strukturu. Frontend si pak data automaticky nalezne sám. Rozměry CSV tabulky streamovaných ADS dat nemusí odpovídat rozměrům výsledného pole — u více než dvourozměrného pole by to ani nebylo možné. Frontend si vytvoří prázdné pole správných rozměrů a do něj sekvenčně vyplňuje přijímaná data. Přidání nového druhu ADS tedy znamená změny v API pouze na straně serveru, načítací service v klientovi se měnit nemusí.

Trochu nešťastné je, že ADS nemají v datastore přiřazený žádný unikátní identifikátor. Jediný způsob, jak konkrétní ADS přesně popsat, je kombinace všech jejích základních atributů. Kvůli tomu některé metody v ADS API vyžadují netriviální množství parametrů, např. metoda pro načítání detailu ADS.

Příklad 1. *Uvažujme situaci, kdy frontend na svůj dotaz na konkrétní ADS dostane tuto odpověď.*

```
{
  type: 'PerNeuronPairValue',
  ids: [3, 4, 5],
  sheet: 'V1_Exc_L4',
  values: {
    '@link': '/ads/pnpv?path=datastore&sheet=V1_Exc_L4',
    dimensions: [3, 3]
  }
}
```

Frontend v tuto chvíli musí získat data z poskytnuté cesty dalším dotazem. Pakliže server vygeneruje následující data:

```
16,7,8
11,2,8
22,9,1
```

Výsledná ADS bude vypadat takto.

```
{
  type: 'PerNeuronPairValue',
  ids: [3, 4, 5],
```

```
  sheet: 'V1_Exc_L4',
  values: [
    [16, 7, 8],
    [11, 2, 8],
    [22, 9, 1]
  ]
}
```

Podrobný popis celého API se nachází v příloze B.

3.2 Webový klient

Klient sestává ze 4 hlavních vizuálních celků. Jedná se o souborový systém, navigátor, inspektor a přehled vybraných neuronů. Inspektor dále dynamicky instancuje modul pro vizualizaci v závislosti na typu vybrané ADS. Skutečná modulární struktura kódu silně vychází z použité technologie, proto bude podrobněji rozebrána v kapitole 4.

Souborový systém, navigátor a inspektor představují postupně specifičtější pohledy do datastore. Při zcela čistém startu klienta je nutné jimi postupně projít, ale jejich stav se zároveň ukládá do URL, takže je možné některé kroky (za předpokladu poskytnutí správných hodnot v URL) při otevření webobé stránky přeskočit.

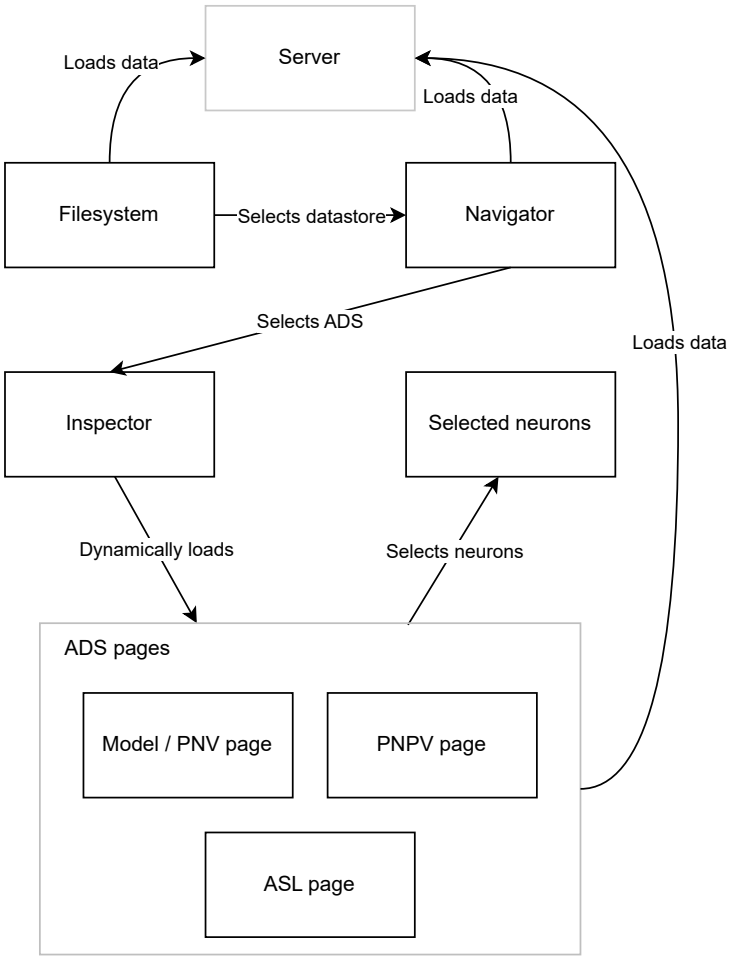
Protože při práci se občas hodí mít před očima co nejvíce informací naráz, a protože kvůli paměťové náročnosti není dobrý nápad mít aplikaci otevřenou naráz ve více záložkách prohlížeče, bylo potřeba najít způsob, jak umožnit změny layoutu a viditelnosti jednotlivých komponent. Zvolili jsme intuitivní způsob, na který už jsou uživatelé zvyklí z jiných míst. V klientovi jsou na několika místech použity kontejnery, zobrazující obsah vedle sebe, kde lze tažením myši měnit rozměry jednotlivých přihrádek. Ukázka na screenshotu 3.3.

3.2.1 Souborový systém

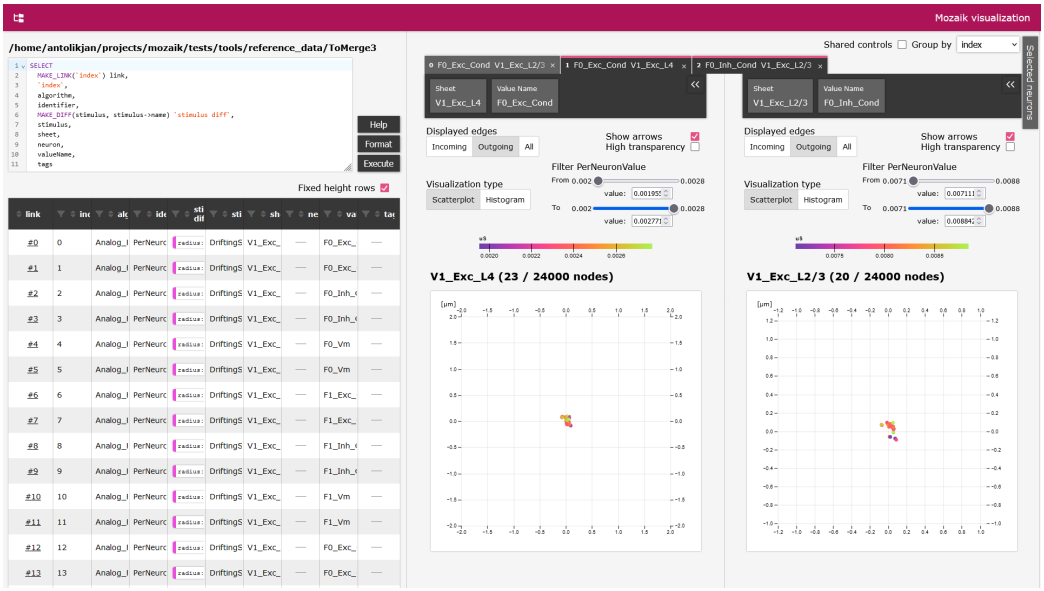
Souborový systém zobrazuje složky a datastory. První složky, které uživatel uvidí, jsou nastaveny v konfiguraci a nelze se z nich dostat ven, pouze zanořovat dovnitř. Je možné rekurzivně prohledat celý adresářový strom a získat přehledně zobrazené pouze cesty, na kterých se nějaký datastore nachází. V určitých případech toto může být příliš pomalé, takže je tu ještě klasický způsob postupného zanořování se do složek. Jakmile uživatel zvolí zamýšlený datastore, klient načte základní data — jednotlivé neuronové vrstvy a spojení a seznam všech ADS.

3.2.2 Navigátor

Navigátor reprezentuje tabulku ADS. Jednotlivé buňky obsahují různé datové typy parametrů, a musí tedy být specializované. Ve stejném sloupci se nicméně



Obrázek 3.2 Architektura frontendu

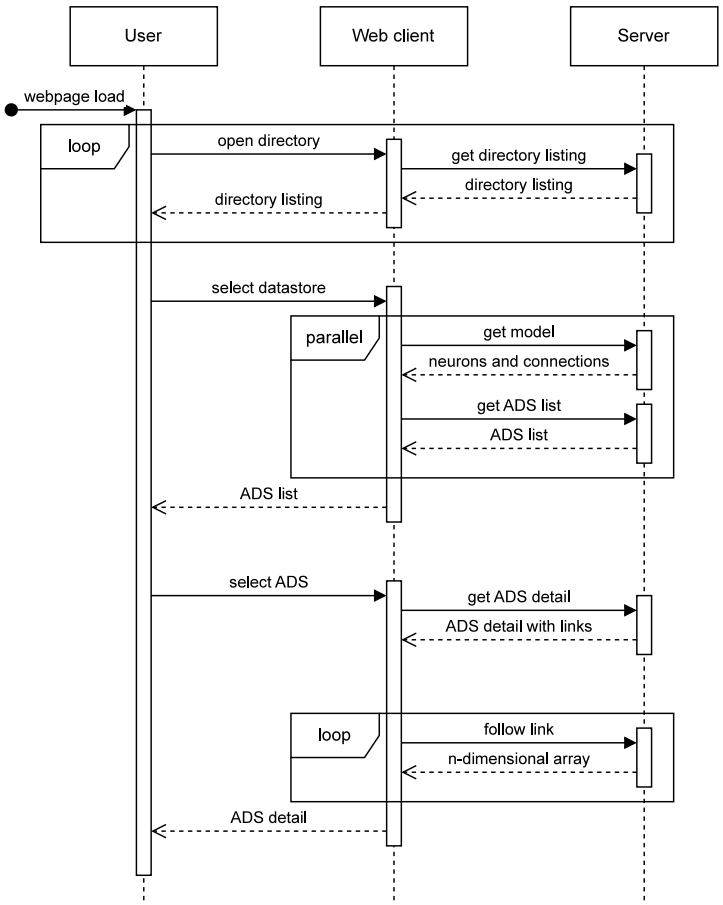


Obrázek 3.3 Ukázka upravovatelného layoutu. Na snímku lze vidět nalevo navigátor a napravo inspektor se dvěma aktuálně zobrazenými záložkami. Je možné myší tahat za hranici mezi navigátorem a inspektorem, za hranici mezi záložkami a za hranice mezi jednotlivými sloupci v navigátoru. Je také možné taháním za pravý okraj aplikace zobrazit sidebar s přehledem vybraných neuronů.

předpokládá stejný datový typ, podobně jako v relačních databázích. Jelikož je potřeba mít opravdu silný a robustní způsob vyhledávání, řazení a projekce ADS, je součástí navigátoru SQL editor. Je možné s ADS pracovat, jako by byly uloženy v relační tabulce s podporou pro datový typ JSON. Kromě standardního SQL jsme přidali několik dalších funkcí, které zjednodušují běžnou práci. Více se o nich opět lze dočíst v kapitole 4. SQL jako součást user experience bylo zvolené s ohledem na to, že očekávanými uživateli jsou vývojáři. Pro zjednodušení ale byly přidány i klasické UX prvky. Sloupce lze jedním kliknutím řadit, lze je také filtrovat pomocí specializovaných dialogů. Interně jsou tyto akce nicméně opět převedeny na SQL query. Tyto pomocné filtry a řazení sice stačí po většinu času, na opravdu komplikované dotazy už ale je potřeba SQL měnit ručně.

3.2.3 Inspektor

Inspektor slouží k prohlížení jednotlivých vizualizací. Mezi požadavky na aplikaci je, že musí být možné vybrat několik ADS a mezi nimi rychle přepínat, nebo jich i zobrazit více vedle sebe. Na takovou činnost jsou uživatelé běžně zvyklí například z prohlížeče nebo textových editorů, které používají záložky. Inspektor proto pro co největší intuitivnost používá totéž řešení. Záložky lze řadit, otvírat, zavírat a kombinovat. Pro každou z nich se potom instancuje modul v závislosti na typu ADS. Za načítání doplňujících dat jsou pak zodpovědné právě tyto specializované



Obrázek 3.4 Příklad komunikace webového klienta a serveru. Po spuštění aplikace je nejprve potřeba vybrat v adresářové struktuře správný datastore. Po jeho zvolení klient načte neuronové vrstvy a jejich spojení a seznam všech ADS. Uživatel si může některou z nich vybrat, tu pak klient načte pomocí jednoho až více dotazů a sestaví kompletní vizualizaci.

moduly.

3.2.4 Přehled vybraných neuronů

Jedná se o komponentu, která zobrazuje seznam právě vybraných neuronů a jejich metadata. Je možné zde výběr i zužovat či rozšiřovat.

Kapitola 4

Implementace

4.1 Server

Serverová část aplikace je napsána v Pythonu. To je jediná rozumná možnost, protože Mozaik je Pythonový framework a bez něj není možné číst data z datastore. Vzhledem k tomu, že potřebujeme jenom jednoduchý lightweight server pro datové API, zvolili jsme framework Flask, který je pro podobné servery populární volbou.

Server je optimalizovaný pro využití jedním uživatelem naráz. To odpovídá předpokládaným scénářům použití, kdy je buď server nasazen na lokálním stroji, nebo využíván malou skupinou výzkumníků. Poslední datastore instance načtená pomocí Mozaiku zůstává cachovaná v paměti dokud není požadován jiný datastore.

4.2 Webový klient

Nejprve je nutné přiblížit si konkrétní použité technologie a jejich principy.

4.2.1 Angular

Webový klient je vyvíjen ve frameworku Angular[1]. Tato volba je podložena řadou argumentů:

1. Angular slouží k vývoji SPA (Single Page Application). To je pro nás klíčová funkce, protože si nemůžeme dovolit opakované načítání stejných dat ze serveru.

2. Angular je aktivně vyvíjen společností Google, která ho sama intenzivně využívá. To je přesvědčivá odpověď na požadavek na framework, který tu ještě nějakou dobu bude.

[1] `https://angular.io/`

3. Angular směřuje vývojáře k vývoji na základě specifických konvencí. Ty vedou k škálovatelnému a udržitelnému kódu. Kromě toho je primárním jazykem Angularu Typescript, který přináší svým statickým typováním stejné výhody.

Komponentová architektura

Angular je založen na komponentové architektuře. Základní stavební blok je komponenta, která se skládá ze tří částí.

Typescriptová třída

Template je psán v jazyce specifickém pro Angular, jenž je nadstavbou nad HTML, která umožňuje do kódu přidávat podmínky, cykly a zobrazovat data typescriptové třídy. Angular sleduje stav aplikace a automaticky podle něj aktualizuje renderované HTML. Za tuto funkčnost je zodpovědný sofistikovaný a vysoce optimalizovaný change detector.

Styly jsou ve výsledném css souboru specifikovány právě pro danou komponentu, takže neovlivňují jiné komponenty.

Kromě komponent sev Angularu používají ještě directives, které nespecifikují vlastní template a styly, ale naváží se na již existující HTML elementy. Příkladem může být například directive, které zvýrazní svůj hostující element při kliknutí. Posledním základním kamenem jsou services, které s výsledným HTML neinteragují vůbec. Services mohou například komunikovat se serverem.

Všechny tři typy tříd mohou využívat dependency injection. Díky tomu je snadné získat například referenci na nějaký důležitý service, nebo na hostující element v případě directive.

Příklad 2. *Příklad Angularové komponenty, která po svém vytvoření načte a zobrazí data ze serveru.*

```
@Component({
  selector: 'test-component',
  template: `
    <span
      class="name"
      *ngFor="let name of names$ | async"
    >
      {{name}}
    <span>
  `
})
class TestComponent implements OnInit {
  names$: Observable<string[]>;

  constructor(private namesS: NamesService) {}
```

```
  ngOnInit() {
    this.names$ = this.namesS.loadData();
  }
}
```

Komponenta získá při vytvoření pomocí dependency injection referenci na service, která umí ze serveru získat seznam jmen. Po jejich načtení pro každé jméno vyrenderuje jeden `span` *element. O tom, co je to* `Observable`*, si povíme za chvíli.*

Tok dat

Komponenty a directives si mezi sebou mohou předávat data buď shora dolu, nebo zezdola nahoru. Shora dolu je to velice jednoduché, zkrátka se v templatu nastaví atribut elementu. Zezdola nahoru se data předávají vyvoláváním eventů.

Příklad 3. *Příklad Angularové komponenty, která komunikuje se svým potomkem*

```
@Component({
  selector: 'child',
  template: ''
})
class Child {
  @Input() propA: number;
  @Output() propB = new EventEmitter<number>();
}

@Component({
  selector: 'parent',
  template: `
    <child
      [propA]="1"
      (propB)="childVal = $event"
    ></child>
  `
})
class Parent {
  childVal: number;
}
```

Komponenta `Parent` *nastaví* `Child` *property* `propA` *na 1. Při vyvolání* `propB` *eventu se spustí handler, který nastaví* `Parent.childVal` *na novou hodnotu.*

Předávání dat na větší vzdálenosti, než je jen rodič a dítě, se dá řešit například pomocí services. Ve větší aplikaci je ale potřeba řídit tok dat systematicky. V Angularu se pro tento úkol používá state management knihovna NgRx. Ta silně využívá (stejně jako Angular) knihovnu RxJS. Nejprve si tedy představíme ji.

4.2.2 RxJS

RxJS[2] je alternativa oproti callbackům nebo promisům využívající principů reaktivního programování. Hlavní roli zde má *observable*, což je objekt, který (většinou asynchronně) produkuje hodnoty. Observables lze všelijak kombinovat, filtrovat, nebo transformovat. Zároveň jsou ve výchozím stavu lazy evaluated.

Příklad 4. *Příklad observable DOM událostí a různých operátorů. Observable začne poslouchat kliknutí na tlačítko až po zavolání* `subscribe`.

```
fromEvent(myButton, 'click').pipe(
  auditTime(100), // ignoruj hodnoty po 100 ms
  // a pak vysli posledni z nich
  take(5), // prvnich 5 hodnot
  filter((event, i) => i % 2 == 0), // sude

  // pro kazdou udalost odesli dotaz
  // na server a dal vysilej data
  // ze serveru (getData vraci Observable)
  mergeMap(() => httpService.getData())
).subscribe(
  data => console.log('server response', data)
);
```

4.2.3 NgRx

Ngrx[3] se stará o globální stav aplikace.

Store

Store je úložiště pro globální stav. Aplikace je o změnách jeho částí notifikována prostřednictvím observables. Stav je imutabilní, musí se vždy nahradit celý. Tento proces se neděje přímočaře, ale pomocí akcí a reducerů.

Action

Action je popis činnosti, kterou se mění stav. V aplikaci je definováno velké množství pojmenovaných akcí. Ty mohou být odkudkoli vyvolány. Akce má jméno a volitelně ještě nějaký payload. Třeba akce `addSelectedNeuron` může mít jako payload id neuronu.

Reducer

Reducer je funkce, která přijme stav a akci a vrátí nový stav. Reducer by měl být deterministický a neměl by mít vedlejší efekty.

[2] `https://rxjs.dev/`
[3] `https://ngrx.io/`

Effect

Vedlejší efekty by měly být definovány zde. Jedná se o třídu, která sleduje akce a na jejich základě něco vykoná a (většinou) vyvolá novou akci.

Příklad 5. *Jak by mohla vypadat sekvence při přihlášení uživatele?*

1. *Komponenta* `LoginDialog` *vyvolá akci* `login({username, password})`
2. *Reducer zpracuje akci a vrátí nový stav, kde* `userLoading = true`
3. `LoginDialog` *je notifikován o novém stavu a zobrazí spinner*
4. *Efekt* `AuthEffects` *si všimne akce a zavolá* `AuthService.verifyUser(username, password)`
5. *Server potvrdí správnost jména a hesla*
6. `AuthEffects` *vyvolá akci* `loggedIn({userData})`
7. *Reducer zpracuje akci a vrátí nový stav, kde* `userLoading = false` *a* `user = userData`
8. `LoginDialog` *je notifikován o novém stavu a zavře se*
9. *Komponenta* `AppShell` *je notifikována o novém stavu a zobrazí uvítací zprávu*

4.2.4 D3.js

D3.js[4] je knihovna pro vizualizaci dat. Její přístup je nízkoúrovňový — místo hotových grafů nabízí dynamický přístup k SVG nebo canvas na základě dat. Poskytuje řadu klíčových prvků, jako jsou interpolátory, osy nebo matematické funkce. Je tedy ideální pro naše účely, kde potřebujeme vizualizace hodně upravovat a přidávat jim interaktivitu. Příklad zde uvádět nebudeme, protože i na obyčejný graf jsou v d3.js potřeba desítky až stovky řádek kódu.

4.2.5 AlaSQL

AlaSQL[5] je in-memory SQL databáze implementovaná v Javascriptu. Podporuje velkou část standardu SQL-99, kromě toho má rozšíření pro práci s JSON nebo grafovými daty. Je to jediný projekt svého druhu a podobného měřítka. Bohužel její architektura není moc propracovaná a kód je špatně udržovatelný [4]. Kvůli tomu je obtížné opravovat její stávající bugy, kterých není málo, a pravděpodobně se nelze vyvarovat bugům budoucím. V aplikaci ale využíváme pouze dotazovací část knihovny, jejíž stav pro naše potřeby dostačuje.

[4] `https://d3js.org/`
[5] `https://github.com/AlaSQL/alasql`

4.2.6 Stav klienta

Veškeré akce, reducery a efekty se nachází ve složce `frontend/src/app/store`. Protože ADS nemají v datastore přiřazený jiný unikátní identifikátor, než je kombinace jejich základních parametrů, frontend je uchovává seřazené a identifikuje je jejich indexy.

UI je část stavu mající na starost uživatelské rozhraní. V tuto chvíli zde jsou jen nastavení pro různé překryvy.

Network drží informace o probíhajících síťových dotazech. Uživatelé mají možnost dotazy vidět a případě zrušit.

Filesystem slouží k uložení adresářové struktury, včetně toho, které složky uživatel rozbalil.

Model je největší část stavu. Uchovává se zde struktura neuronové sítě, tj. neurony a jejich pozice v různých vrstvách a jejich odchozí spojení. Kromě toho se zde drží seznam vybraných neuronů, neuron, na kterém se zrovna nachází kurzor (ten se pak v aplikaci zvýrazní na všech místech, kde je vidět) a případně stav načítání modelu ze serveru.

ADS je seznam všech ADS (zde jsou jen jejich základní informace), seznam načtených ADS (se všemi daty) a případně, podobně jako u modelu, stav jejich načítání ze serveru.

Navigator obsahuje aktuální SQL dotaz.

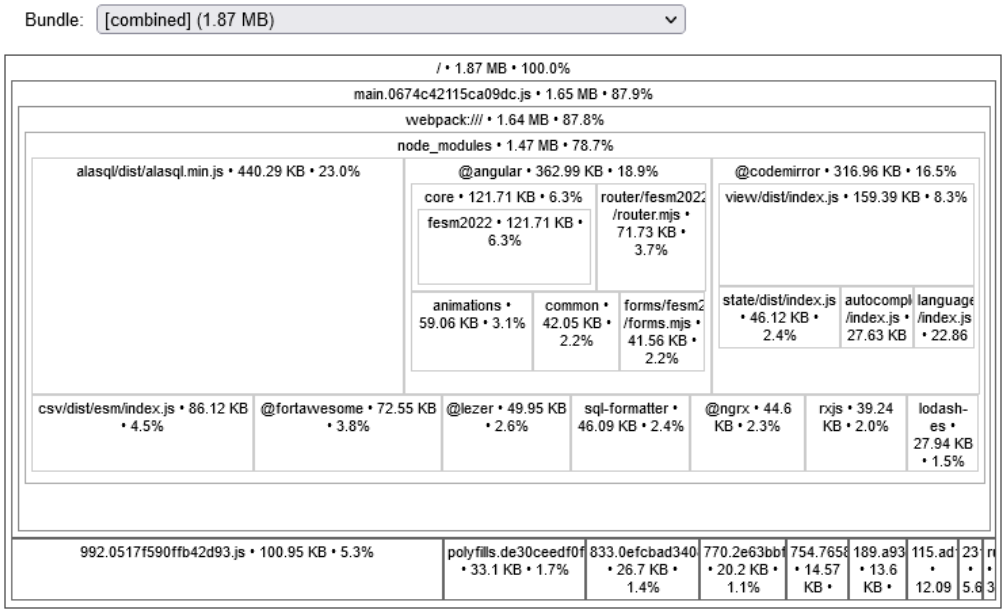
Inspector drží stavy jednotlivých záložek. Ty si každý vizualizační modul definuje sám a slouží většinou k uložení nastavení vizualizace. Kromě toho se zde ukládá, zda současně zobrazené vizualizace stejného typu sdílí svá nastavení.

Následující části globálního stavu se nachází v URL:

- cesta k aktuálnímu datastore
- indexy aktuálně načtených ADS
- indexy aktuálně zobrazených ADS
- atribut, podle kterého jsou seskupované zobrazené ADS

4.2.7 Moduly

Kód je rozdělen do několika Angular modulů podle toho, kde všude je potřeba jej využívat.



Obrázek 4.1 Rozložení minifikovaného kódu napříč webovým klientem. Je vidět, že většinu objemu tvoří knihovny třetích stran. Velikost jednotlivých dynamicky importovaných modulů (vespod vpravo) se pohybuje mezi 1.1% až 0.02%

Common

Toto je modul pro všechny komponenty, které jsou za běhu aplikace vždy zobrazeny. Patří sem tabulka navigátoru, záložková struktura inspektoru, záhlaví stránky a podobně.

Widgets

Kolekce často používaných komponent. Většina z nich jsou různé formulářové prvky, například input, range, nebo checkbox. Dále tu je stránkovač, podpora pro toast notifikace a kontejnery s měnitelnými rozměry.

DSPages

Jedná se o složku modulů pro vizualizaci každé podporované ADS. Tyto moduly jsou načteny dynamicky za běhu podle toho, jaká ADS je zobrazena. Díky tomu je zmenšený objem Javascriptu, který klient musí před svým spuštěním načíst. Jak je ale vidět na obrázku 4.1, nejde o zásadní rozdíl. Sdílené části všech vizualizací se nachází v modulu `DSPagesCommon`.

4.2.8 SQL

Pro snazší filtrování a transformace jsme do SQL přidali následující nestandardní funkce.

`MAKE_LINK(value)`

Funkce očekává index ADS a vrací objekt, který se ve výsledné tabulce zobrazí jako kliknutelný odkaz na přidání ADS do inspektoru.

`JSON_STRINGIFY(value)`, `JSON_PARSE(value)`

Jedná se o funkce pro převod hodnot mezi JS objekty a deterministickým JSON. Hodí se například pro `GROUP BY`, protože AlaSQL objekty porovnává podle reference.

`MAKE_INTERSECTION(value)`

Tato agregační funkce vrací průnik properties se stejným jménem i hodnotou ve všech poskytnutých objektech.

Příklad 6. *Ukázka použití* `MAKE_INTERSECTION` *na object a na pole.*

```
-- data: [
--   {a: {foo: 'bar', x: 1}},
--   {a: {foo: 'bar', x: 2}},
-- ]

SELECT MAKE_INTERSECTION(a) a FROM data

-- vysledek: [
--   {a: {foo: 'bar'}},
-- ]

-- data2: [
--   {a: [1, 2, 7]},
--   {a: [3, 7, 1, 6]},
-- ]

SELECT MAKE_INTERSECTION(a) a FROM data2

-- vysledek: [
--   {a: [1, 7]},
-- ]
```

SUBTRACT(minuend, subtrahend)

Fuknce, která vrací množinový rozdíl mezi první a druhou hodnotou.

Příklad 7. *Ukázka použití* `SUBTRACT` *na object a na pole.*

```
SELECT SUBTRACT(
  @{foo: 1, bar: 2, baz: 3},
  @{foo: 2, bar: 2, car: 3}
) a

-- vysledek: [
--   {a: {foo: 1, baz: 3}},
-- ]

SELECT SUBTRACT(
  @[1, 2, 3],
  @[4, 3, 7]
) a

-- vysledek: [
--   {a: [1, 2]},
-- ]
```

`MAKE_DIFF(value, diffId = 0)`

Tato funkce není ani klasická funkce, ani agregátor. Je implementována pomocí `MAKE_INTERSECTION` a `SUBTRACT`, ale část z ní se musí vykonat zcela mimo AlaSQL. Více implementačních detailů je v sekcích 5.2.10 a 5.2.5.

Schová atributy se stejným jménem a hodnotou ve všech poskytnutých objektech. Je možné poskytnuté objekty oddělit pomocí `diffId`. Tato funkce se používá například ve výchozím výpisu ADS, kde se zobrazují odlišné hodnoty ve stimulech se stejným jménem (`MAKE_DIFF(stimulus, stimulus->name)`).

Příklad 8. *Ukázka použití* `MAKE_DIFF` *na object a na pole.*

```
-- data: [
--   {a: {foo: 'bar', x: 1}},
--   {a: {foo: 'bar', x: 2}},
-- ]

SELECT MAKE_DIFF(a) a FROM data

-- vraci: [
--   {a: {x: 1}},
--   {a: {x: 2}},
-- ]

-- data2: [
--   {a: [1, 2, 7]},
--   {a: [3, 7, 1, 6]},
-- ]

SELECT MAKE_DIFF(a) a FROM data2

-- vraci: [
--   {a: [2]},
```

```
--   {a: [3, 6]},
-- ]

-- data3: [
--   {a: {foo: 'bar', x: 1}},
--   {a: {foo: 'bar', x: 2}},
--   {a: {foo: 'cow', x: 1}},
--   {a: {foo: 'cow', x: 2}},
-- ]

SELECT MAKE_DIFF(a, a->foo) a FROM data3

-- vraci: [
--   {a: {x: 1}},
--   {a: {x: 2}},
--   {a: {x: 1}},
--   {a: {x: 2}},
-- ]
```

Kapitola 5

Bližší pohled na frontend

V této kapitole se podrobněji podíváme na implementaci klíčových tříd ve webovém klientu.

5.1 Widgets

5.1.1 MultiviewComponent

Tato komponenta je zodpovědná za layout s rozšiřovatelnými sloupci nebo řádky. Rozlišujeme hlavní a vedlejší osu. Ve směru hlavní osy lze rozšiřovat jednotlivé přihrádky, ve směru obou os lze přihrádky řadit. Ve výchozím stavu je hlavní osa horizontální. Vedlejší osa se projeví ve chvíli, kdy je specifikována seskupovací funkce. Ta seskupí přihrádky ve směru kolmém na osu hlavní a nadále se k nim chová jako k celku.

Veřejné API

mainAxisOrder, secondaryAxisOrder jsou reference na funkce, které specifikují řazení přihrádek. Ve výchozím stavu jsou přihrádky řazeny podle pořadí jejich vytvoření.

secondaryAxisGroup je reference na seskupovací funkci. Ve výchozím stavu přihrádky seskupované nejsou.

ratios, ratiosChange dohromady tvoří bidirectional binding, tj. je to kombinace property a event emitteru. Jedná se o poměry velikostí jednotlivých přihrádek.

relativeRatios udává, zda je velikost přihrádek specifikována v procentech, nebo v pixelech. Pokud v pixelech a poměry nejsou definované, využije se property `defaultGroupSize`.

vertical je property nastavující směr hlavní osy.

Jednotlivé přihrádky jsou v templatu specifikovány pomocí komponenty `MultiviewPartitionComponent`. Ta se nevyrenderuje, ale poskytne rodičovské `MultiviewComponent` referenci na svůj obsah. Každá přihrádka má property `data`, která se využívá pro řazení a seskupování. Kromě toho má event emitter `visible`, který signalizuje, zda má přihrádka vyšší než minimální rozměr. Minimální rozměr je zde proto, aby i v zcela zavřeném stavu bylo možné přihrádku myší snadno uchopit a roztáhnout.

Příklad 9. *Základní zobrazení pomocí* `MultiviewComponent`. *Výsledný layout by měl mít 3 sloupce.*

```
@Component({
  template: `
    <mozaik-multiview-component
      [mainAxisOrder]="order"
      [secondaryAxisGroup]="group"
    >
      <mozaik-multiview-partition
        *ngFor="let i of numbers"
        [data]="i"
      >
        <span>number {{ i }}</span>
      </mozaik-multiview-partition>
    </mozaik-multiview-component>
  `
}) class Usage {
  numbers = [1, 15, 7, 8];
  order = (a, b) => a - b;
  group = (x) => x % 3;
}
```

5.1.2 DraggableDirective

Toto directive se aplikuje na elementy, u kterých má uživatel mít možnost myší měnit pořadí. V aplikaci pomocí něj byly implementovány záložky pro ADS. Elementy jsou přehazovány uvnitř svého rodičovského elementu a pouze horizontálně. Po uchopení myší se element vyjme ze svého layoutu a na jeho místo se umístí stejně velký placeholder, který je možné stylovat. Vyjmutý element se pak posouvá s kurzorem. Pakliže by měl svým prostředkem přesáhnout prostředek vedlejšího elementu, vedlejší element a placeholder si animovaně prohodí pozice. Po puštění kurzoru je placeholder nahrazen zvednutým elementem. Pokud je rodičovský element horizontálně scrollovatelný a zvednutý element je přesunut myší úplně na kraj viditelného úseku, ale ještě není na kraji skutečného obsahu, rodičovský element bude scrollovat tak, aby bylo možné pokračovat v prohazování jeho dětí.

Veřejné API

`DraggableDirective` nabízí 4 eventy.

- swapLeft
- swapRight
- lift
- drop

Poznámka. Na této třídě lze obzvlášť ocenit eleganci, jakou nám nabízí RxJS. Podívejme se, jak stručně se dá shrnout hlavní funkčnost.

```
fromEvent<MouseEvent>(
  this.elRef.nativeElement,
  'mousedown'
)
  .pipe(
    tap((e) => {
      e.preventDefault();
      this.lift();
      const el = this.elRef.nativeElement;
      this.startX = e.clientX;
      this.startOffsetX = el.offsetLeft -
        el.parentElement.scrollLeft;
    }),
    switchMap(() =>
      this.gEventS.mouseMove.pipe(
        switchMap((e) => this.scrollIfNeccessary(e)),
        takeUntil(
          this.gEventS.mouseReleased.pipe(
            tap(() => {
              this.drop();
            })
          )
        )
      )
    )
  )
  .subscribe((e) => {
    this.drag(e);
  });
```

5.1.3 PropertyBagComponent

Jde o komponentu pro vizualizaci key-value objektů. Jednotlivé properties jsou zobrazeny jako samostatné elementy ve formě `key: value`, přičemž po snazší vizuální orientaci má každý takový element přiřazenou barvu vygenerovanou ze jména property. Komponenta může fungovat i jako radio input pro výběr property. Toho je využito v navigátoru ve filtru pro key-value datové položky.

Veřejné API

Komponenta implementuje `ControlValueAccessor` interface, takže je schopna fungovat v Angular formulářích. Její radio funkčnost se zapne, když je do nějakého takového formuláře zapojena.

bag data, které chceme vizualizovat

5.2 Common

5.2.1 FilesystemComponent

Sidebar, který zobrazuje strom adresářů a datastore. Rekurzivně zobrazuje `FolderComponent` a `DatastoreComponent`. `DatastoreComponent` se chová jako odkaz, který do URL přidá cestu ke správnému datastore. `FolderComponent` vysílá akce `loadDirectory`, `loadRecursiveFilesystem` a `toggleDirectory` (ta mění viditelnost obsahu složky).

Veřejné API

files objekt popisující adresářovou strukturu

5.2.2 SelectedNeuronsComponent

V této komponentě uživatel může zkoumat neurony, které vybral v inspektoru. Pro každý neuron se vypisují metadata a jeho spojení, vše v rozbalovacích sekcích. Vycházející hrany se zobrazí rovnou, protože jsou ve stavu aplikace u každého neuronu uloženy. Pro vstupní hrany je ale potřeba projít celou síť, což u velké sítě a mnoha vybraných neuronů trvá dlouho. Proto se tato akce odkládá, dokud uživatel poprvé nerozbalí dotyčnou sekci u daného neuronu.

Ať je neuron zobrazen v komponentě kdekoli (tj. ať ve své sekci, nebo jako druhý konec hrany), po najetí kurzorem se vyvolá `hoverNode` akce. Při kliknutí se vyvolá buď `selectNodes` nebo `addSelectedNodes`, podle toho, zda se klikne se shiftem. Zároveň komponenta čte ze stavu, jaký neuron je právě pod kurzorem, a zvýrazní jej na všech jeho pozicích.

Komponenta nemá žádné veřejné API, protože komunikuje přímo se store pomocí akcí.

5.2.3 DsTabsComponent

Jedná se o komponentu, která na základě stavu URL vytváří komponenty pro vizualizace viditelných ADS. Komponenta pro vizualizaci se zničí a vytvoří znovu kdykoli se přepne záložka na jinou a nazpátek. Každá taková komponenta zároveň implementuje interface `DSPage`, která obsahuje referenci na template jménem

`controls`. Když je zapnuté sdílení nastavení, `DsTabsComponent` zobrazí v k tomu určeném místě `controls` z první viditelné ADS vizualizace každého typu. Součástí template `DsTabsComponent` jsou samotné záložky, které jsou implementovány pomocí `DraggableDirective`. Komponenta naslouchá jejich událostem a na základě toho aktulizuje URL, čímž se mění pořadí renderovaných vizualizací. Rovněž zajišťuje formulářový prvek pro výběr atributu, podle kterého jsou seskupovány zobrazené ADS v `MultiviewComponent`.

5.2.4 DSTabHandleComponent

Komponenta záložky. Zajišťuje především vizuální stránku, ale zároveň mění URL při kliknutí na sebe různými tlačítky myši. Tím buď zobrazuje ADS, kterou reprezentuje, nebo ji zcela zavírá.

Veřejné API

ds podmnožina parametrů zobrazované ADS. Očekává se, že budou zobrazovány pouze parametry odlišné od ostatních připravených ADS.

viewing je input, který nastavuje, zda je záložka prohlížena, nebo jen připravena.

ignoreClick je technický input obsluhovaný rodičovskou komponentou. Při tažení myší je totiž prohlížečem vyvolána událost kliknutí, který potřebujeme vyfiltrovat pryč.

Příklad 10. *Ukázka použití* `ignoreClick`. *Příklad je převzat ze zdrojového kódu* `DsTabsComponent`.

```
<mozaik-ds-tab-handle
  #handle
  mozaikDraggable
  *ngFor="let item of ready$ | async"
  [ds]="item.ds"
  [viewing]="item.viewing"
  (lift)="handle.ignoreClick = false;
    initReorderedTabs()"
  (swapLeft)="handle.ignoreClick = true;
    swapTabs(item.ds, true)"
  (swapRight)="handle.ignoreClick = true;
    swapTabs(item.ds, false)"
  (drop)="commitReoderedTabs()"
></mozaik-ds-tab-handle>
```

5.2.5 DSSelectComponent

Hlavní komponenta celého navigátoru. Její zodpovědností je propojit SQL editor a tabulku filtrovaných ADS. Zde se také počítá `MAKE_DIFF` SQL funkce (4.2.8). Při

každé změně query (to nemusí být jen v editoru, ale i pomocí např. filtrovacích dialogů) se pomocí AlaSQL získá seznam filtrovaných a transformovaných ADS. Komponenta zkontroluje, zda byla `MAKE_DIFF` v query využita a pokud ano, pro každou datovou buňku zjistí, zda má asociovaná `MAKE_DIFF` metadata. Pro každý sloupec, ve kterém se metadata vyskytují, spočítá pomocí `MAKE_INTERSECTION` (4.2.8) průnik pro každou hodnotu `diffId`. Tu pak od relevantních datových buněk odečte pomocí `SUBTRACT` (4.2.8).

5.2.6 SQEEditorComponent

Tato komponenta je wrapper nad editorem zdrojového kódu CodeMirror[1].

Veřejné API

content modifikovatelný obsah editoru

query event emitter pro spuštění dotazu

error event emitter notifikující o chybách při formátování

5.2.7 DSTableComponent

Zde se zobrazuje SQL výstup. Při změně dat komponenta nejprve určí unikátní hodnoty v každém sloupci. Následně z první neprázdné hodnoty v každém sloupci určí jeho datový typ. Pro každý sloupec pak data umístí do správného typu buňky. Mapování datových typů na typy komponent pro buňky je nakonfigurováno v hashmapě v typescriptové třídě `DSTableComponent` a v templatu se tento typ komponenty určuje dynamicky. Kvůli tomu není možné poskytnout datové buňce hodnotu přímo pomocí input atributu, ale je potřeba ji vložit pomocí dependency injection.

Příklad 11. *Dynamický typ datové buňky. V template použitá pipe* `purefn` *slouží pouze k optimalizaci počtu volání.*

Template

```
<ng-template
  *ngComponentOutlet="
    cellTypes[colSchema[key]];
    injector: getInjector | purefn :
      row[i] :
      rowRef.injector
  "
></ng-template>
```

[1] `https://codemirror.net/`

funkce getInjector

```
getInjector(value: any, parent: Injector) {
  return Injector.create({
    providers: [{
      provide: DSCELL_VAL,
      useValue: value
    }],
    parent,
  });
}
```

konstruktor konkrétní datové buňky

```
constructor(@Inject(DSCELL_VAL) public value: any) {}
```

Veřejné API

src dvourozměrné pole dat

keys pole názvů jednotlivých sloupců

rowHeight input pro vynucení výšky řádku. Hodí se, pokud některé hodnoty jsou moc rozsáhlé a neúměrně zvyšují celý svůj řádek. Buňka, která je v takovém případě moc velká, dostane scrollbar.

5.2.8 CellHeaderComponent

Tato komponenta slouží jako záhlaví pro sloupec v `DSTableComponent`. Na základě typu buňky otevírá specifický filtrovací dialog (což je poměrně běžná komponenta, která pomocí formulářových prvků generuje SQL podmínky). Umí také vyvolat akce `sortByColumn` a `addCondition`.

Veřejné API

key název sloupce

type datový typ sloupce

distinctValues unikátní hodnoty ve sloupci. Pokud jich je dostatečně málo, filtrovací dialog z nich dá uživateli přímo na výběr.

5.2.9 SQLBuilder

Tato třída je klíčová pro modifikaci SQL dotazů. Využívá toho, že AlaSQL poskytuje přístup k vygenerovanému AST (Abstract Syntax Tree) pro každou query. AST není zdokumentovaný a jeho `toString` metoda nefunguje správně, proto bylo nutné nejprve naprogramovat vlastní převod AST na dotaz. Strukturu AST jsme

reverse-engineerovali a pro každou z nich byla napsána specifická metoda. Není to ideální návrh, ale modifikovat dynamicky neznámé třídy by bylo horší.

Příklad 12. *Ukázka fluent syntaxe* `SQLBuilder`

```
SQLBuilderFactory
  .create('SELECT a, b FROM data')
  .andWhere('a > b')
  .andWhere('a > 5')
  .wrap()
  .orderBy({key: 'a', asc: true})
  .toString();
// SELECT * FROM (
//   SELECT a, b FROM data WHERE
//   a > b AND a > 5
// )
// ORDER BY a ASC
```

Veřejné API

statements počet statementů ve zpracovávaném dotazu

orderBy přidá k dotazu řazení podle zadaného sloupce. Nahradí veškeré dosavadní řazení

andHaving přidá `AND` podmínku do `HAVING` části dotazu

andWhere přidá `AND` podmínku do `HAVING` části dotazu

wrap udělá z aktuálního dotazu subquery

extractSortColumn vrací sloupec, podle kterého se dotaz primárně řadí

getColumnNames vrací jména sloupců

getAggregatedCols vrací jména sloupců, které náleží k agregovacím funkcím

escapeColumn v poskytnutém SQL výrazu ošetří jména sloupců. Využívá přitom znalosti jmen sloupců v současném dotazu.

escapeValue ošetří hodnotu libovolného typu pro použití v SQL dotazu

sanitizeStr ošetří speciální znaky v textu

toString získá textový dotaz

toFormattedString získá hezky formátovaný textový dotaz

5.2.10 makeDiff

Implementace SQL `MAKE_DIFF` funkce. Vrací kopii hodnoty s přidanými metadaty. Skutečné filtrování properties se děje v této komponentě: 5.2.5.

5.3 DSPagesCommon

5.3.1 DSPage

Abstraktní třída, která vyvolává akci `loadSpecificAds` a poskytuje svým potomkům přístup k výsledné kompletní ADS. Dále jim poskytuje přístup k jejich části globálního stavu vizualizací.

5.3.2 HistogramComponent

Komponenta zodpovědná za renderování histogramu. Na výsledný histogram zobrazuje medián a aritmetický průměr hodnot. Počet přihrádek v histogramu lze do určité míry ovlivnit — d3, které přihrádky počítá, akceptuje doporučený počet přihrádek.

Veřejné API

data číselná data, ze kterých je histogram vytvořený

extent input pro rozsah číselných hodnot. Je poskytnut zvenku, protože se ve vizualizaci většinou používá na víc místech a dává tedy smysl ho počítat pro všechny využití naráz.

thresholds doporučený počet přihrádek, nebo funkce, která tento počet spočítá z dat

5.3.3 ZoomFeature

Funkčnost pro vykreslení číselných os do 2D grafu, interaktivní přibližování a pohyb.

Veřejné API

transform get-only property obsahující popis současné transformace souřadnic

zoomCallback se poskytne v konstruktoru a spustí se při každé transformaci

5.3.4 NetworkGraph

Vizualizace neuronové vrstvy ve formě scatterplotu. Po označení neuronu umí zobrazit hrany vedoucí ven nebo dovnitř. Vysílá akce `hoverNode`, `selectNodes` a `addSelectedNodes`.

Veřejné API

nodes seznam neuronů k zobrazení

allNodes seznam všech neuronů

sheetName jméno zobrazované vrstvy

showArrows zda zobrazovat šipky pro jednotlivá neuronová spojení. Občas se hodí vypnout, je-li vrstva hodně hustá.

highTransparency se rovněž hodí použít při velmi hustých vizualizacích. Nevybrané neurony jsou vždy částečně průhledné, ale s tímhle nastavením výrazně víc.

edgeDir směr zobrazovaných spojení. Buď vstupní, výstupní, nebo obojí.

5.3.5 LassoFeature

Funkčnost pro `NetworkGraph`, která umožňuje myší nakreslit cestu a vybrat ohraničené neurony.

Veřejné API

rebind metoda pro vyznačení prvků, které je možné lasovat

5.3.6 NetworkZoomFeature

Třída dědící od `ZoomFeature`, která navíc vykresluje čtvercovou síť.

Veřejné API

transformX, transformY transformuje souřadnice neuronu do souřadnic grafu

5.4 ModelPage

5.4.1 ModelPageComponent

Dostali jsme se k první dynamicky importované vizualizační komponentě. Tato komponenta umí vizualizovat jak Connections ADS, tak PerNeuronValue, jejíž

vizualizace je jen nástavbou na Connections. V obou případech je hlavní částí scatterplot s neurony, ale PerNeuronValue je navíc obarví podle jejich přiřazených hodnot. PerNeuronValue je kromě toho možné vizualizovat jako histogram. S tím vším se pojí některá další možná nastavení vizualizace, kde změny některých hodnot je časově náročné zpracovat (jmenovitě jde o filtr viditelných hodnot a doporučený počet přihrádek v histogramu). Proto při změně pouze těchto náročných parametrů, které jsou navíc z uživatelského hlediska implementované jako slidery, se vysílání `setTabState` akcí odkládá, dokud neuběhlo od poslední změny alespoň 100ms.

5.4.2 PNVNetworkGraphComponent

Podtřída `NetworkGraphComponent`, která přidává podporu pro obarvování neuronů a jejich filtrování.

Veřejné API

pnv hodnoty jednotlivých neuronů

pnvFilter filtr zobrazených hodnot

5.4.3 PNVFeature

Poskytuje `PNVNetworkGraphComponent` funcionalitu pro filtrování hodnot a barvení neuronů

Veřejné API

pnvLength počet hodnot, které jsou vidět za daného filtru

getColor barva pro daný neuron

setData poskytnutí `PNVFeature` filtru a hodnot neuronů

filterPNV schová odfiltrované neurony

filterEdges schová hrany, které vedou do nebo z odfiltrovaného neuronu

redraw obarví neurony

5.5 PerNeuronPairValue

5.5.1 PerNeuronPairValuePageComponent

Slouží k vizualizaci PerNeuronPairValue. Umí ji zobrazit jak jako histogram, tak jako matici. Podobně jako `ModelPageComponent` umí filtrovat hodnoty a nastavovat

doporučený počet přihrádek v histogramu, takže i tato komponenta se snaží brzdit rychle se měnící hodnoty daných nastavení.

5.5.2 MatrixComponent

Toto je dotyčná matice. Původní plán byl renderovat celou matici pixel po pixelu v SVG, ale jednotlivých čtverečků rychle bylo tolik, že i na hodně výkonném počítači byla její responzivita nepřípustná. Pixely se proto kreslí na canvas a jeho obsah se pak zobrazuje jako SVG image element. Omezíme se tím o některé vizuální efekty a musíme dopočítavat, nad kterým čtverečkem se nachází kurzor, ale namísto toho je vizualizace příjemná na používání.

Veřejné API

matrix vstupní data

filter filtr hodnot

periodic zda jsou hodnoty v matici periodické. V takovém případě se používá jiná barevná škála.

unit jednotka hodnot v matici

5.6 ASLPage

5.6.1 ASLPageComponent

Tato komponenta zobrazuje AnalogSignalList, tedy seznam záznamů hodnoty měnící se s časem, kde každý záznam je přiřazený k jednomu neuronu. Ve vizualizaci je `NetworkGraphComponent`, který slouží pro výběr neuronů. Vybrané neurony pak mají svou hodnotu vizualizovanou v podobě lomené čáry ve vedlejším grafu.

5.6.2 LinesGraphComponent

Komponenta pro vykreslení závislosti veličiny na čase. Jedná se o čárový graf, který lze interaktivně přibližovat a posouvat ve směru osy x. Podporuje vykreslování více veličin naráz, přičemž každé z nich přiřazuje barvu podle jejího pořadí. Kdykoli dojde ke změně zobrazovaných veličin, komponenta spočítá Voroného diagramy z poskytnutých bodů. Při pohybu myší nad grafem se pak zvýrazňuje nejbližší bod a lomená čára, jejíž je součástí. Zároveň je vyslána akce `hoverNode` pro neuron náležející dané veličině. Voroného diagramy se počítají pro souřadnice při největším možném přiblížení grafu a při pohybu myší se pozice kurzoru do těchto souřadnic transformuje. Zobrazovaných bodů může být velké množství a bez tohoto opatření mohou být diagramy při velkém přiblížení kvůli zaokrouhlovacím chybám nepřesné.

Veřejné API

ds vizualizovaný AnalogSignalList

selected seznam označených neuronů

Kapitola 6

Návod k použití

6.1 Instalace

Nejprve je nutné nainstalovat runtime pro Python a Javascript.

- Node.js 18
- Python 3.10

Pak je potřeba nainstalovat Mozaik. Přesný postup je uveden v jeho dokumentaci[5].

Dále doinstalujeme pip a npm balíčky.

```
pip install -r requirements.txt
npm i -g @angular/cli
cd frontend
npm i
```

Ve stejné složce frontend zkompilujeme.

```
npm run build
```

6.2 Spuštění

Pro snadné spouštění je v kořenovém adresáři připraven skript `run.sh`. Má několik volitelných parametrů.

```
Start up flask server for mozaik visualization
Usage: ./run.sh [options]

Options:
    --expose
        listen on all network interfaces (default false)
    -h, --help
        display this message
    --port
```

```
    port to listen on (default 5000)
--prod
    use production parameters (default false)
--restart
    restart on crash (default false)
--root
    root folder for looking up datastores,
    can be specified multiple times (default .)
```

6.3 Testy

Testy lze zvlášť spouštět pro Python a pro Angular.

```
pytest
cd frontend
npm run test
```

Angularové testy ze své podstaty musí běžet v prohlížeči. Adresa k otevření je vypsaná ve výstupu testovacího příkazu.

6.4 Dev server

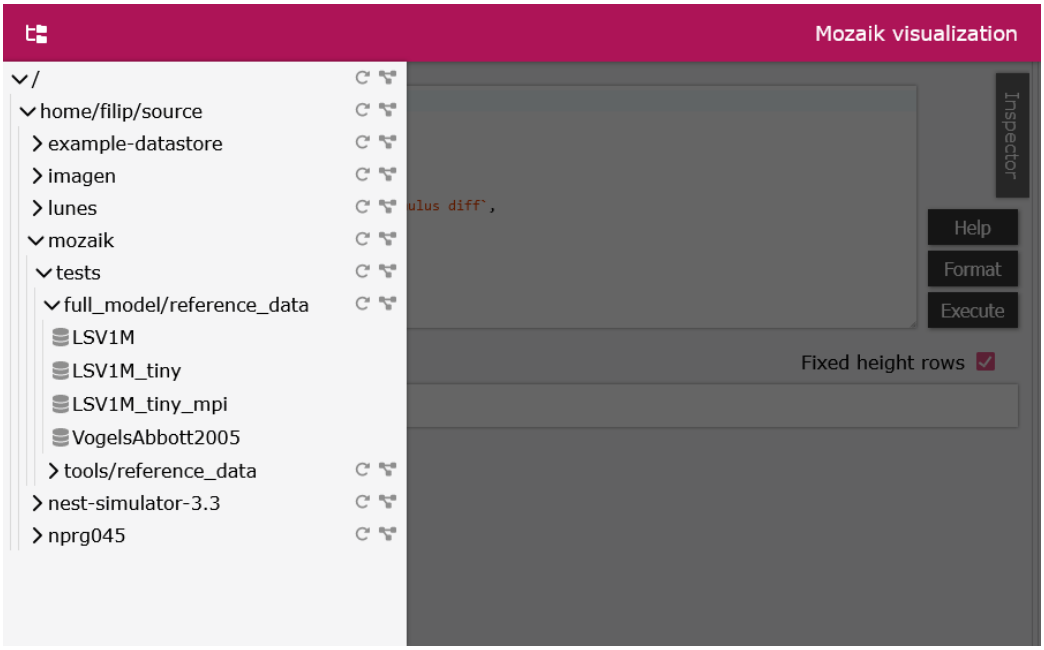
Při vývoji Angularové části je doporučeno používat Angular dev server. Spustí se následujícím příkazem.

```
cd frontend
npm run start
```

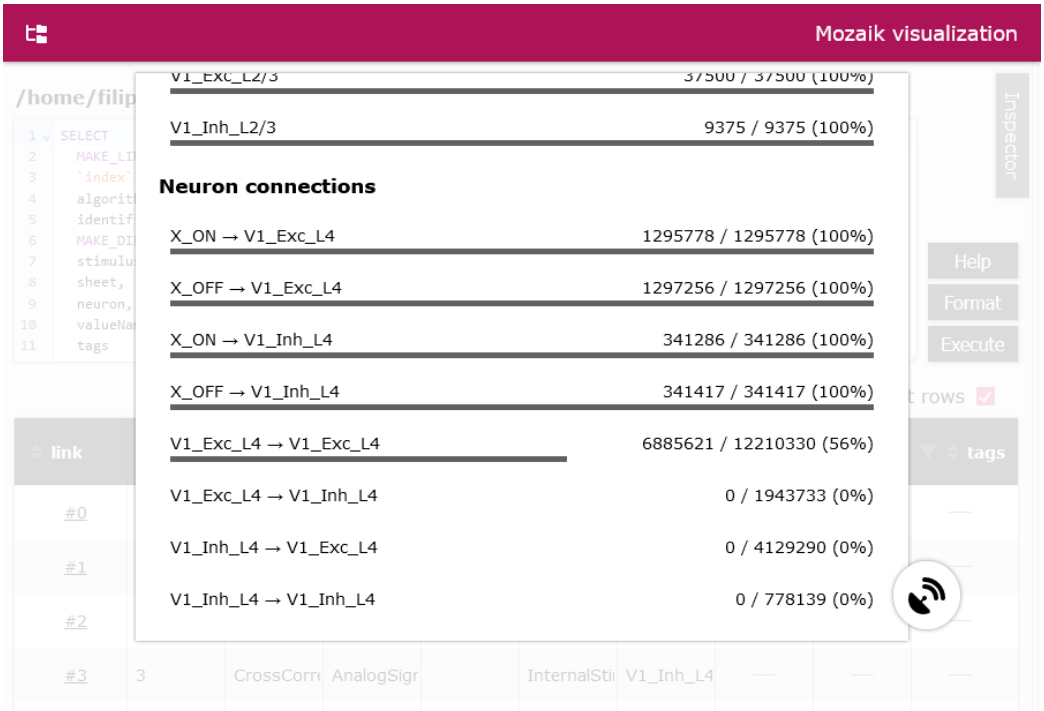
Tento server generuje sourcemaps pro snazší debugování, má zapnutou podporu pro store devtools, rekompiluje se při změnách kódu a automaticky přenačte otevřenou stránku. I nadále je ovšem potřeba mít spuštěný Flask server pro poskytnutí API.

6.5 Základní práce s programem

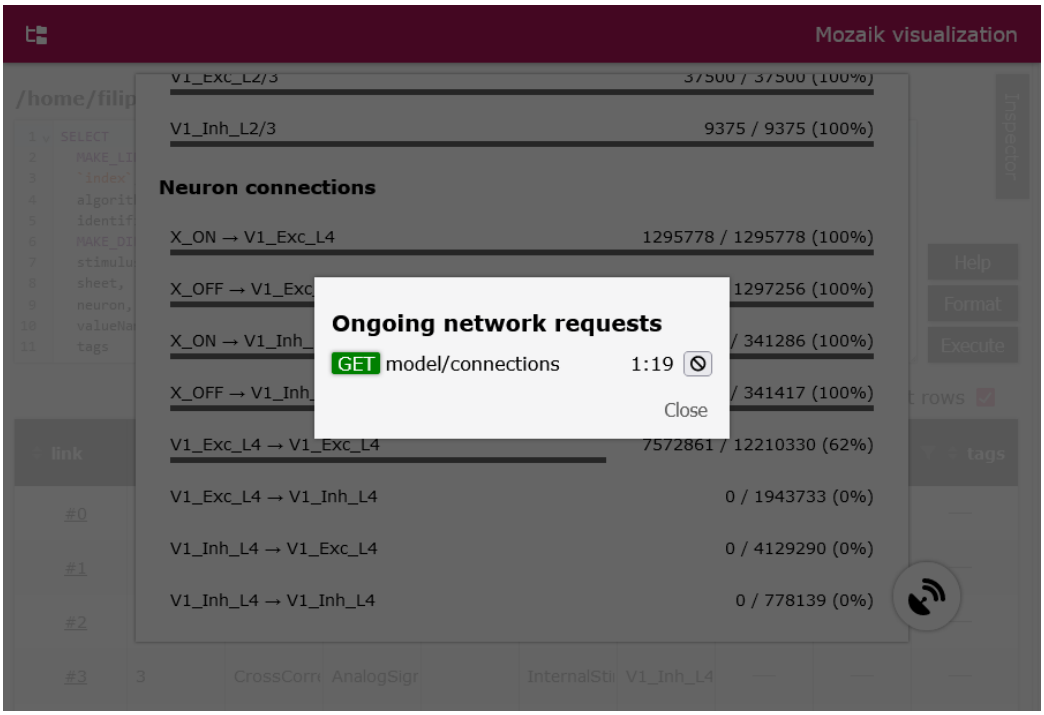
Protože program má grafické rozhraní, bylo by poněkud nešikovné bavit se o jeho používání jenom v textu. Následující část tedy bude sekvence obrázků, které nás provedou od načtení stránky po jednotlivé vizualizace. Aby byly dobře vidět, jsou některé screenshoty pořízeny na malé obrazovce. Při standardním běhu je na všechno více místa.



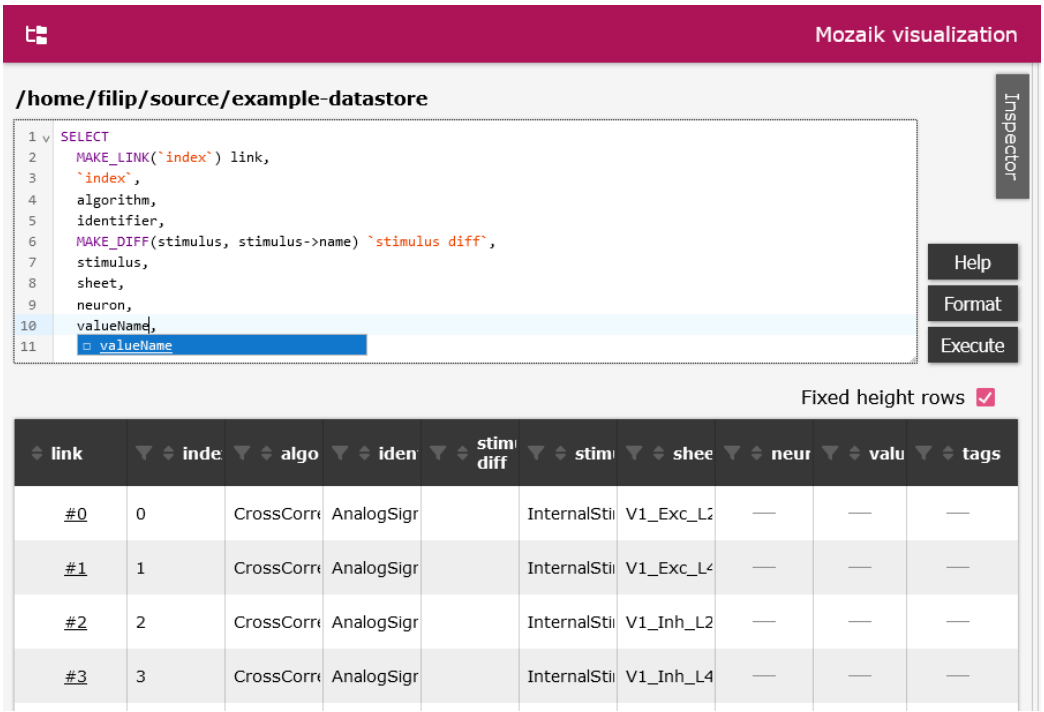
Obrázek 6.1 Po načtení programu se zobrazí prohlížeč adresáře. Před další prací je nutno najít a vybrat datastore. Jednotlivé složky je možné procházet. Každá složka má navíc dvě tlačítka vpravo — jedno z nich přenačte obsah složky a druhé její obsah načte rekurzivně a vyfiltruje složky bez datastore. Rekurzivně načtená byla složka `mozaik`, proto jsou názvy některých složek v ní ve skutečnosti delší cesty.



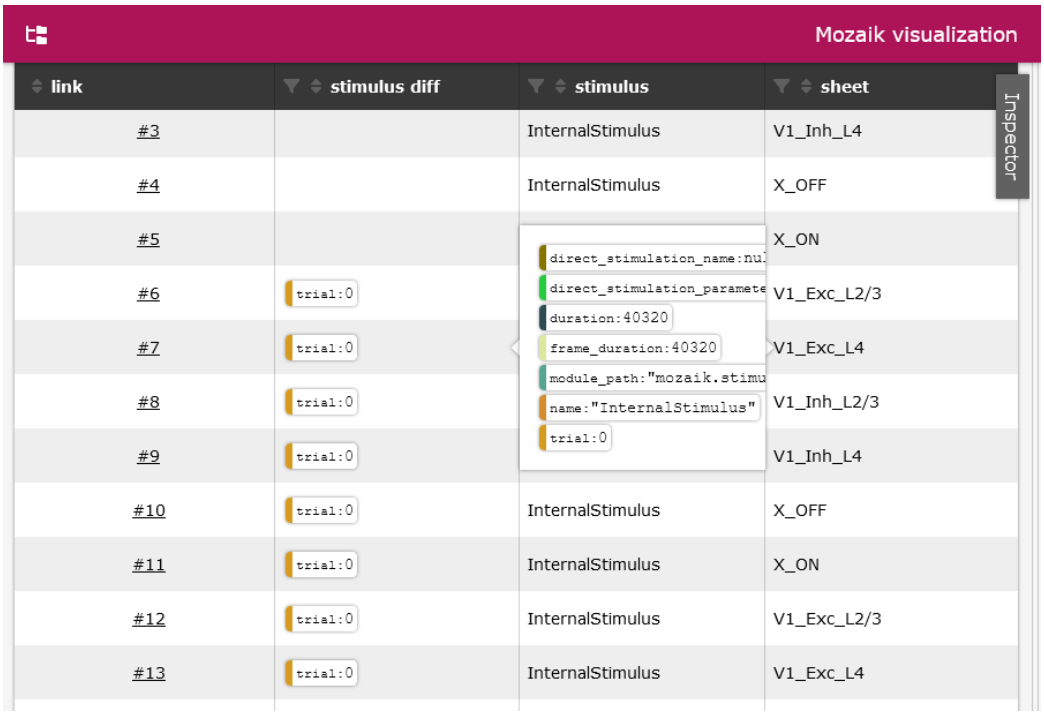
Obrázek 6.2 Jakmile je datastore zvolen, načte se model. Při jeho načítání je vidět, kolik dat už se stáhlo. Vpravo dole je tlačítko, které zobrazí probíhající síťové dotazy — to se zobrazí při jakékoli komunikaci po síti.



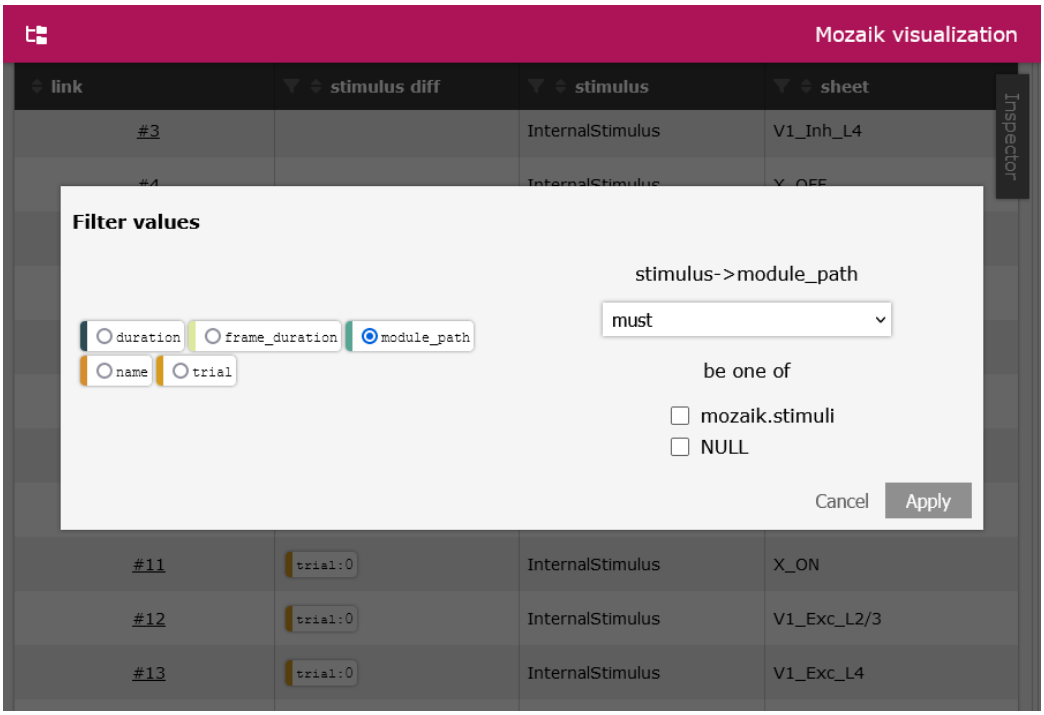
Obrázek 6.3 V tomto dialogu jsou vypsané jednotlivé probíhající dotazy. Dotaz může selhat a být spuštěn znovu, v takovém případě se zde ukazuje i o kolikátý pokus jde. Po kliknutí na tlačítko lze dotaz zrušit. To je vhodné v případě, že nejde o první pokus a dotaz zahrnuje hodně dat. Může to totiž znamenat, že byl server přetížen, došla mu paměť a byl mezitím restartován. Takový dotaz by akorát způsobil další restart serveru.



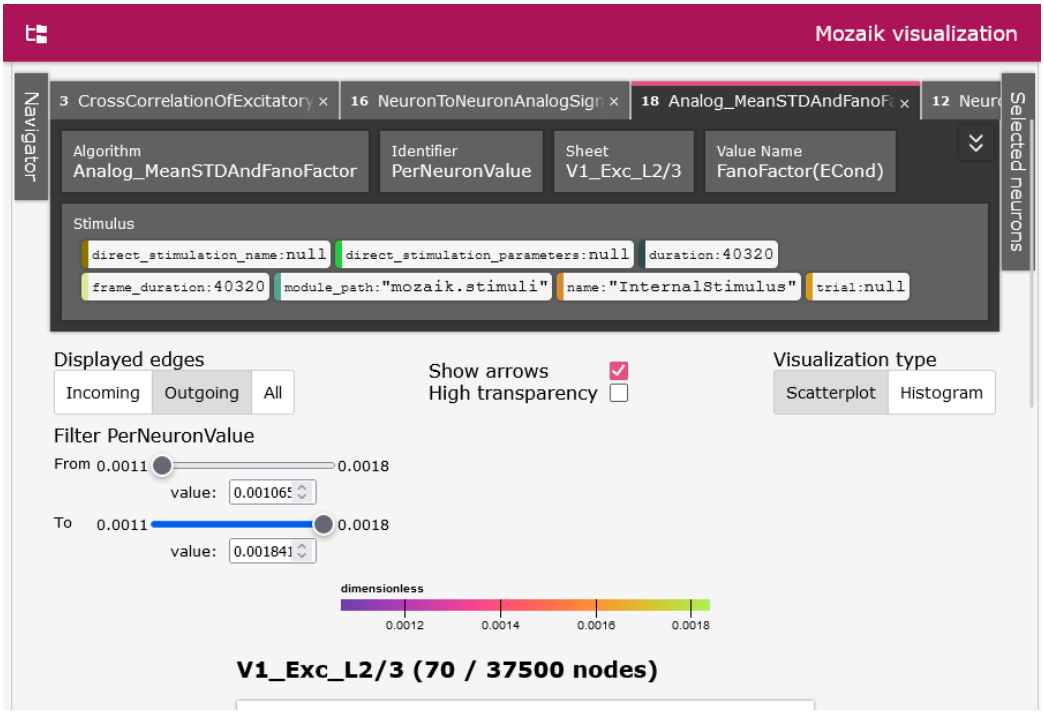
Obrázek 6.4 Všechny ADS jsou načteny do in-memory SQL tabulky, kterou je možné dotazovat. Po kliknutí na název sloupce je možné řádky seřadit. Vedle názvu je také vidět tlačítko pro zobrazení filtrů. V záhlaví celé stránky je nalevo vidět tlačítko pro otevření výběru datastore. To je dostupné neustále.



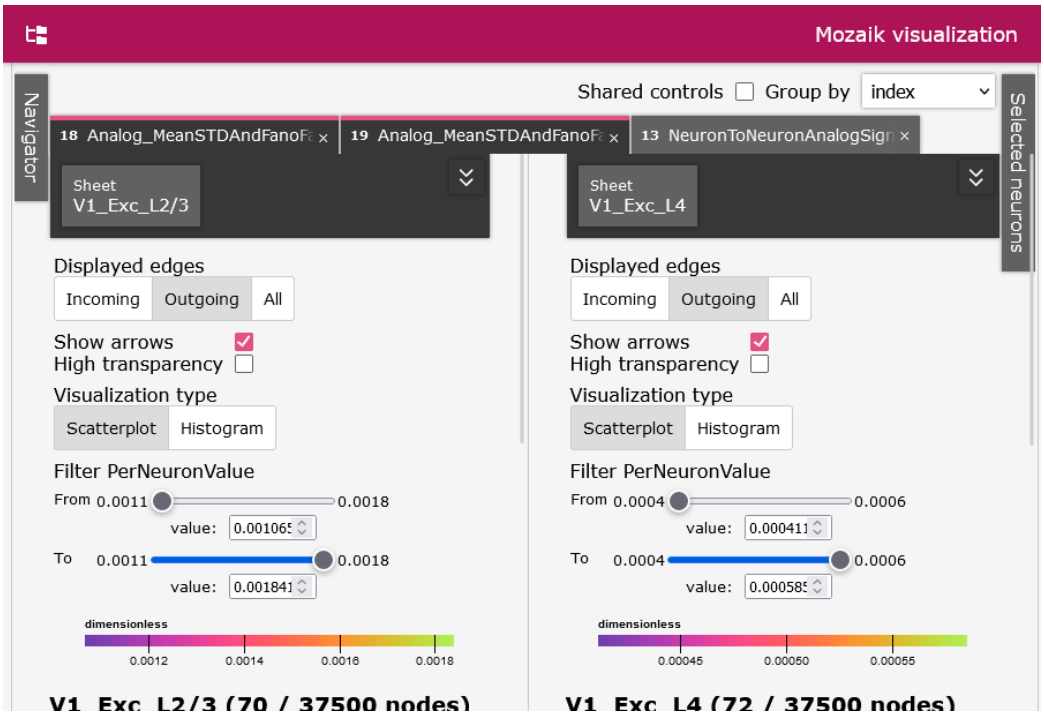
Obrázek 6.5 Stimulus má většinou příliš mnoho parametrů na to, aby se do tabulky příjemně vešel. Zobrazuje se tedy jen jeho jméno a parametry jsou přístupné po najetí kurzorem.



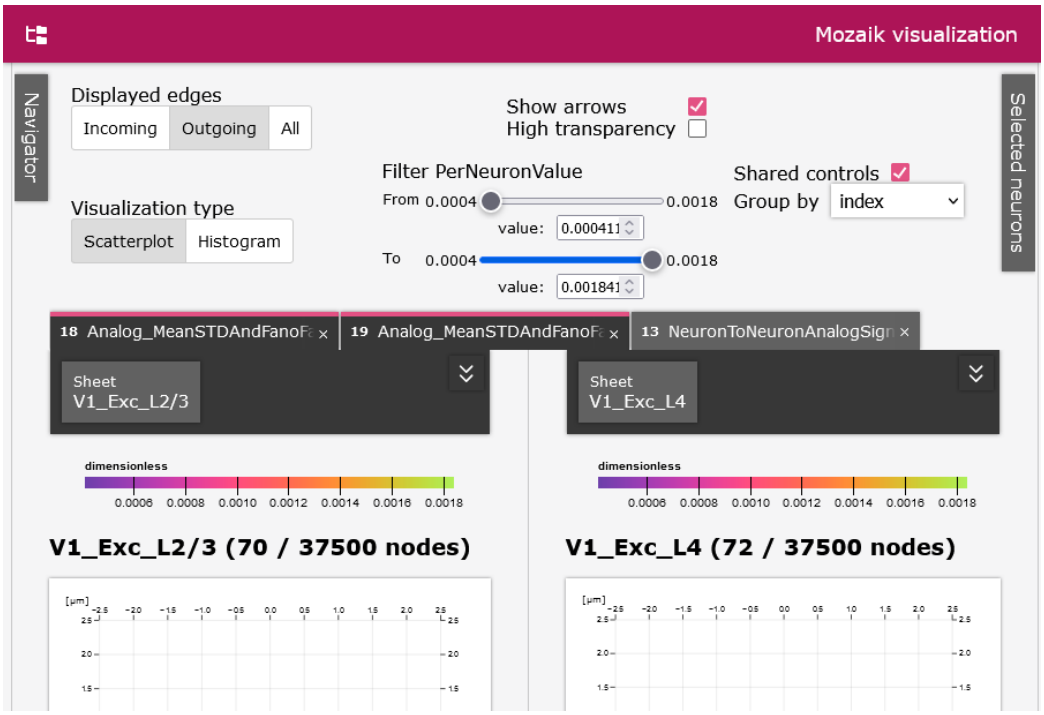
Obrázek 6.6 Takto vypadá filtr sloupce s key-value hodnotami. Položka `module_path` má mezi všemi stimuly jen dvě hodnoty, takže není třeba komplikovanějších podmínek.



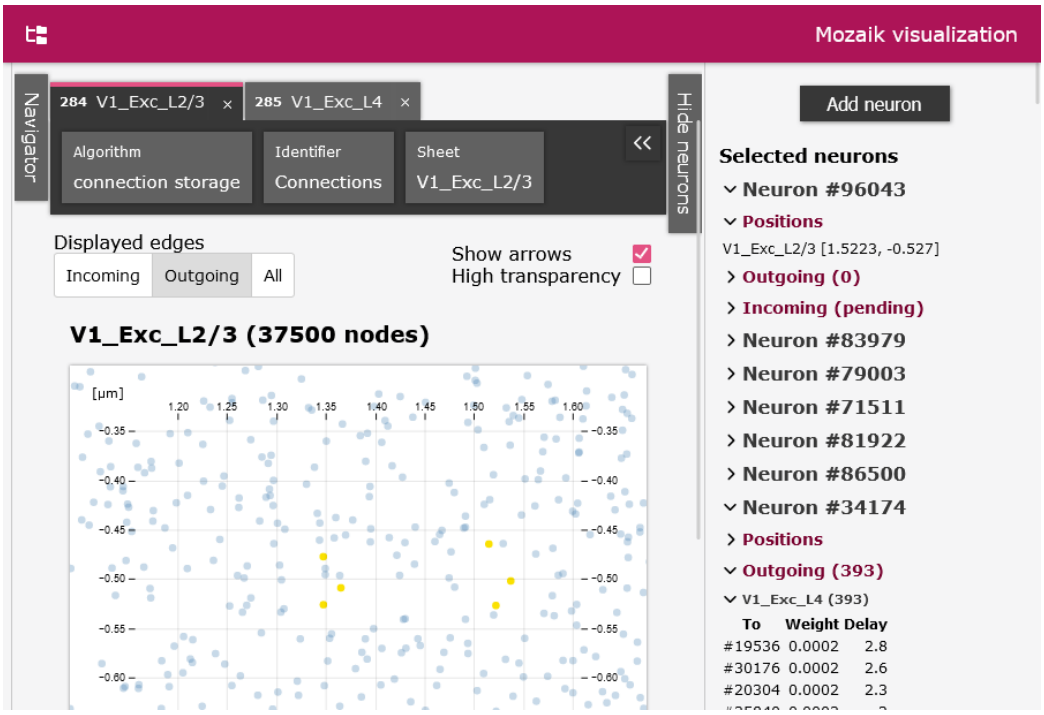
Obrázek 6.7 Takto vypadá inspektor jednotlivých ADS. Jména záložek jsou tvořena na základě odlišných parametrů.



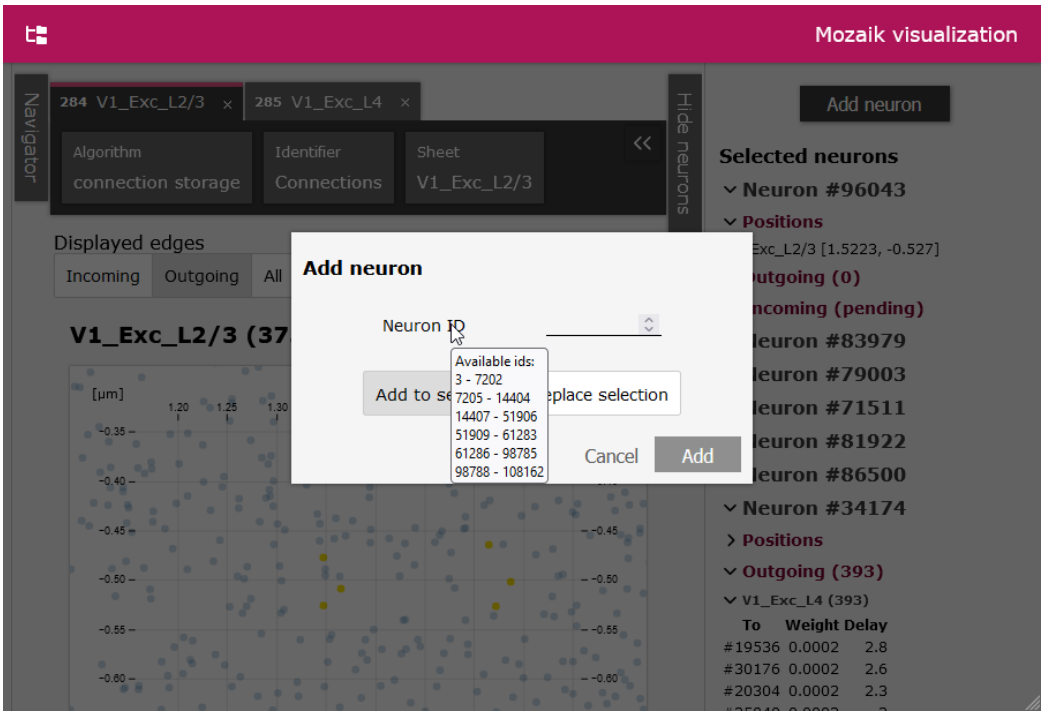
Obrázek 6.8 Když se vybere víc záložek naráz, základní informace o ADS ve tmavém bloku jsou redukovány na rozdílné položky.



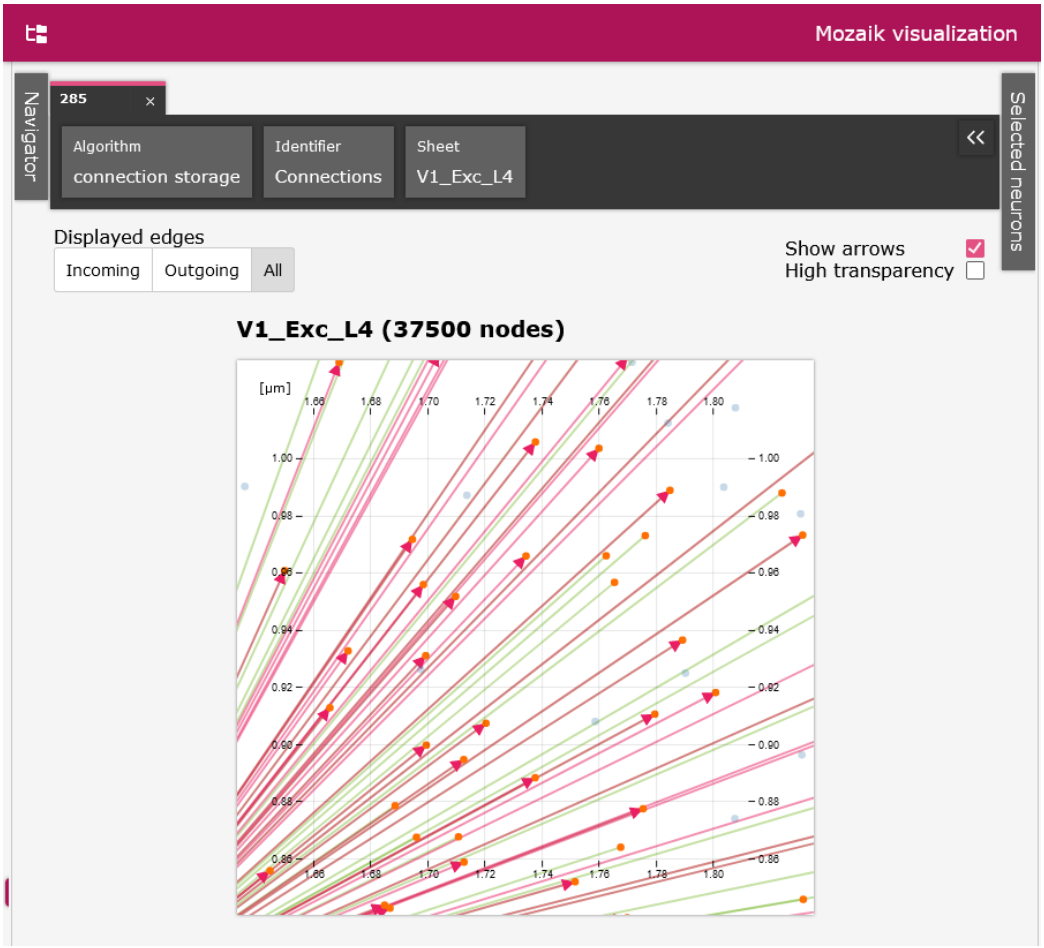
Obrázek 6.9 Když jsou sdílená nastavení záložek, přesouvají se nahoru. Je vidět, jak se oproti 6.8 propojilo minimum a maximum ve sliderech uprostřed.



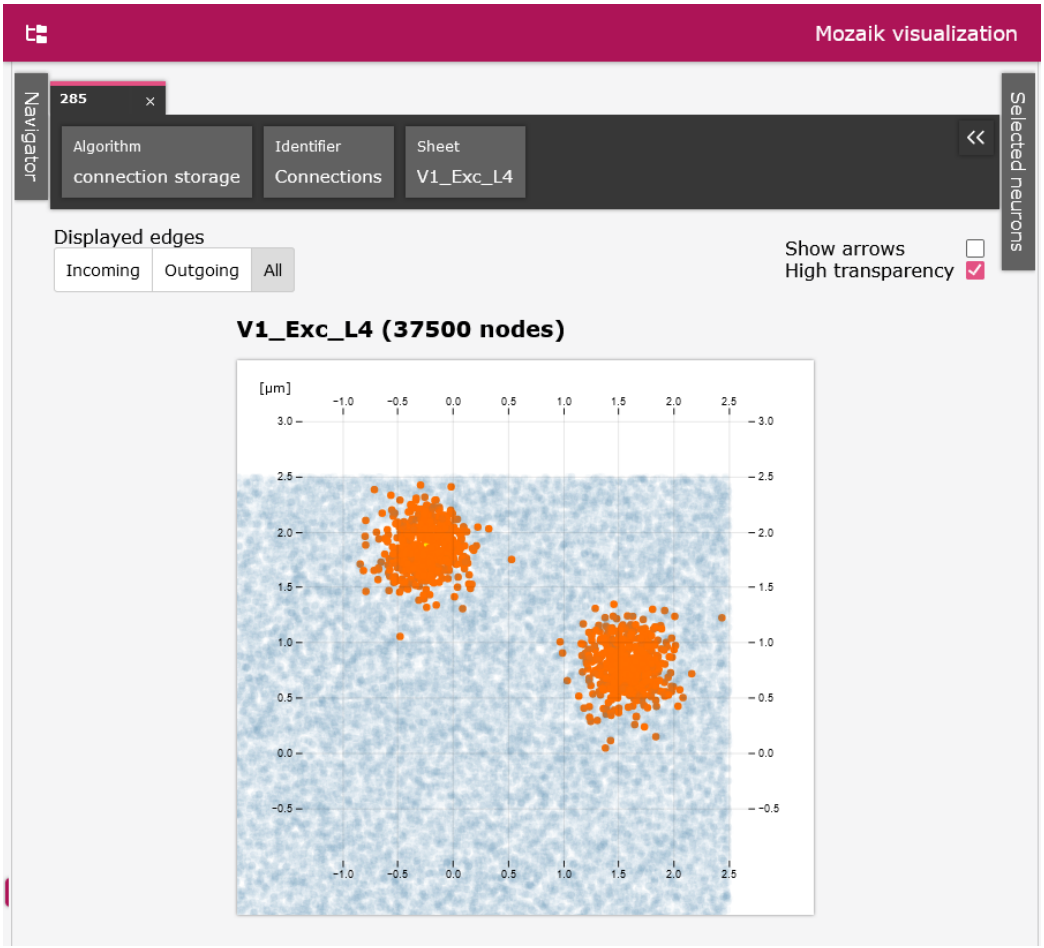
Obrázek 6.10 Když jsou vybrané nějaké neurony, vypisují se ve speciální sekci. Vstupní spojení se spočítají až po rozbalení, protože jde o náročnou operaci. Při najetí myší na libovolný neuron se neuron zvýrazní na všech místech, kde je vidět, včetně grafu.



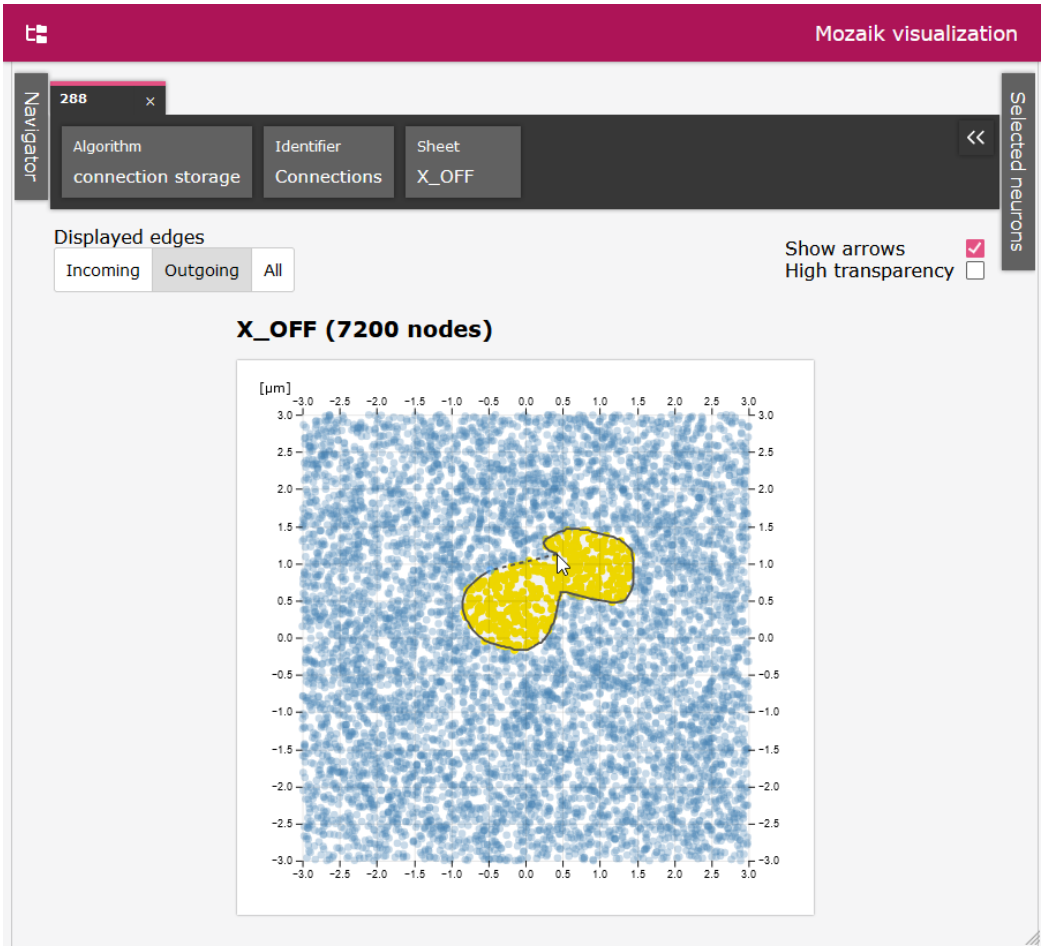
Obrázek 6.11 Neurony lze vybírat buď v grafech, nebo v tabulce spojení vypsané v sekci vybraných neuronů, nebo pomocí dialogu.



Obrázek 6.12 Takto vypadá vizualizace spojení v neuronové vrstvě. Červené šipky vedou ven z vybraného neuronu a zelené dovnitř.

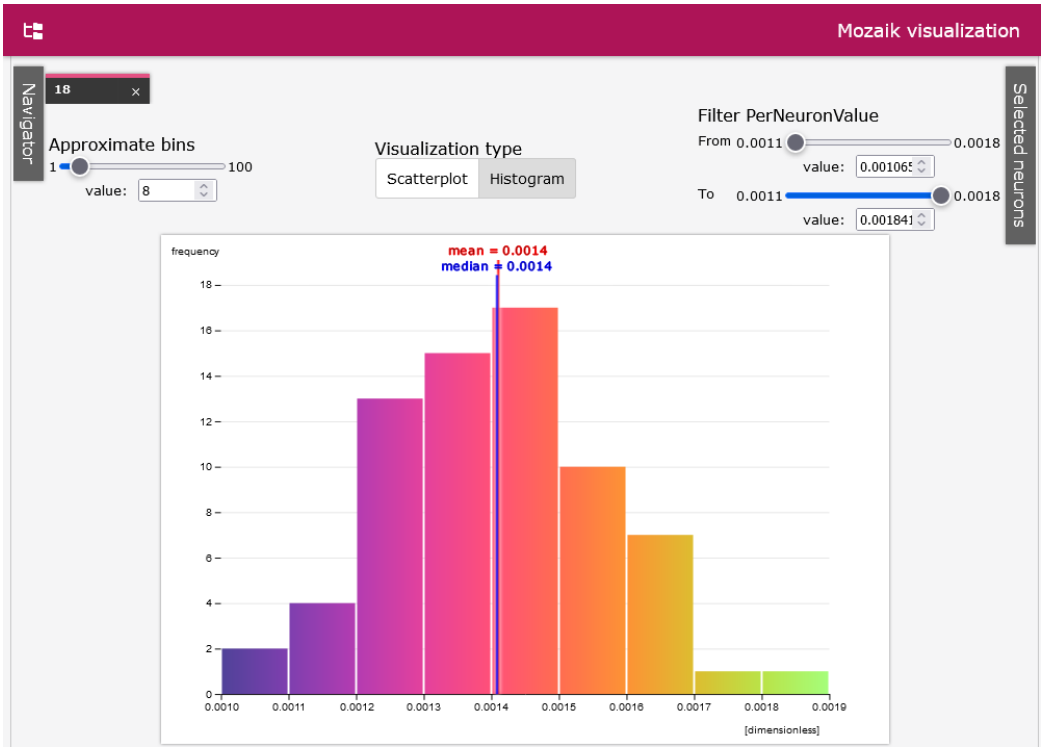


Obrázek 6.13 Když je neuronů příliš mnoho, spojení jsou špatně vidět. V nastavení vizualizace lze vypnout šipky mezi neurony a zvýšit průhlednost nevybraným.

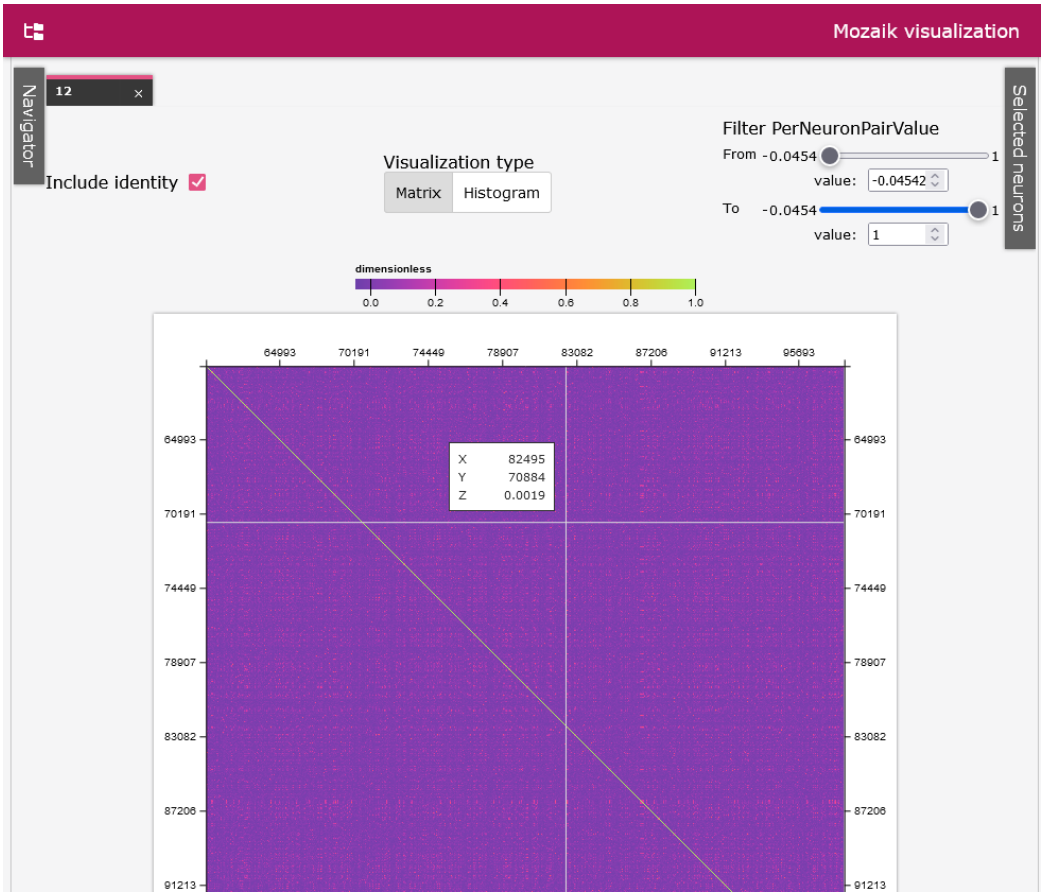


Obrázek 6.14 Neuron lze vybrat kliknutím. Kliknutí s klávesou shift neuron do výběru přidá nebo odebere, zachovávajíc ostatní vybrané neurony. Je také možné s klávesou shift opsat myší cestu a vybrat všechny neurony uvnitř. Klávesou escape se pak výběr zruší.

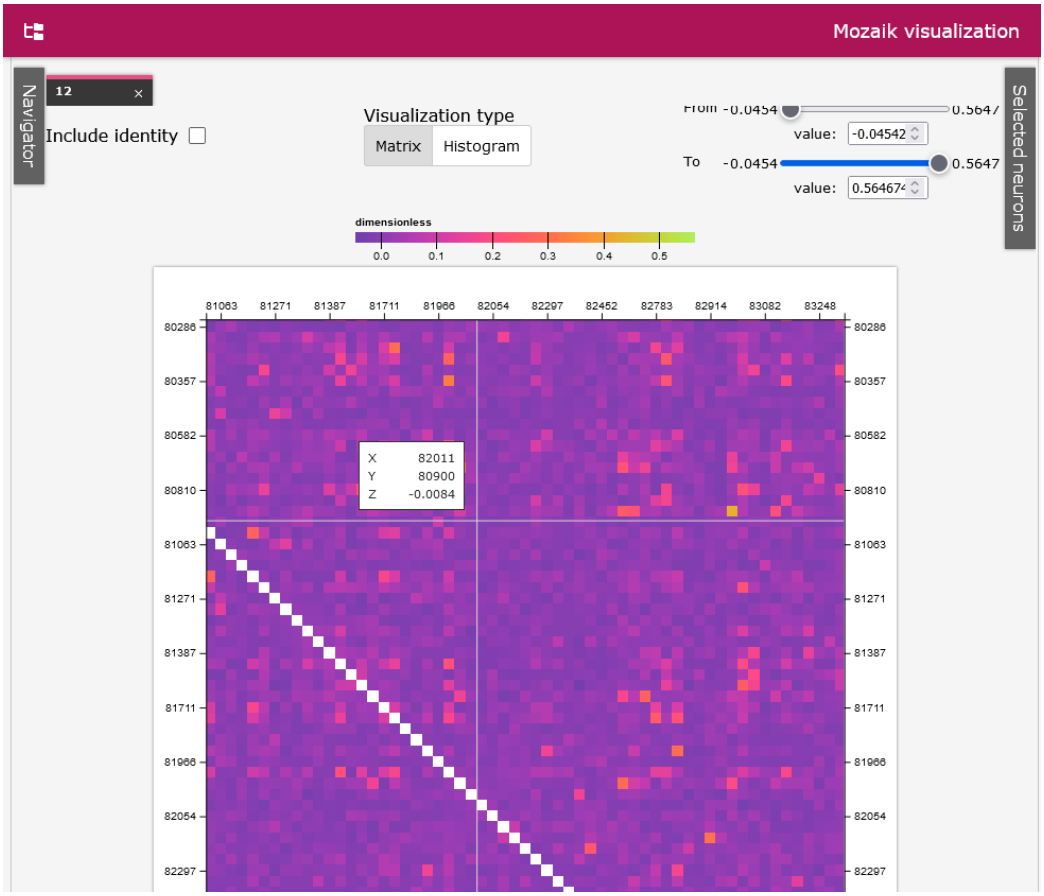
Obrázek 6.15 PerNeuronValue ADS je vizualizovaná jako obarvený graf neuronové vrstvy. Kdyby šlo o periodickou veličinu, barevná škála by byla cyklická. Je možné filtrovat neurony na základě jejich hodnoty.



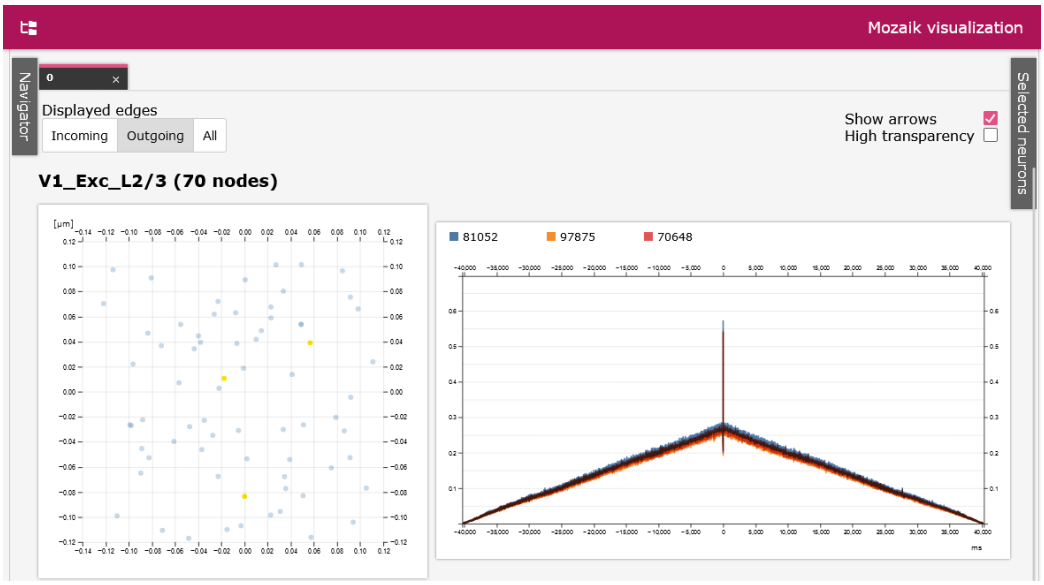
Obrázek 6.16 Alternativní zobrazení PerNeuronValue je v podobě histogramu. Je možné do určité míry ovlivnit počet přihrádek pomocí slideru nahoře vlevo. Histogram ukazuje také medián a aritmetický průměr hodnot.



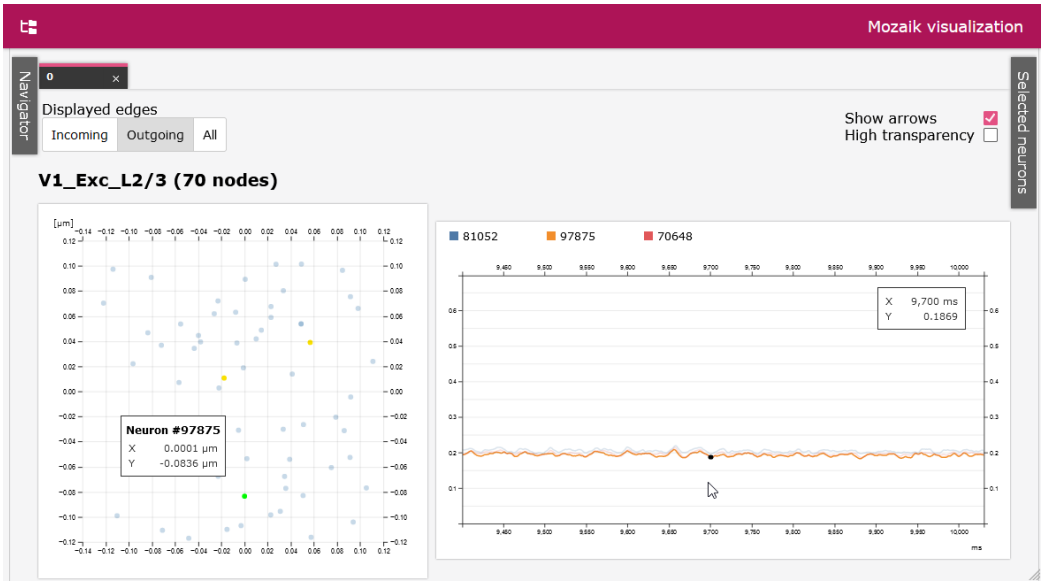
Obrázek 6.17 PerNeuronPairValue je vizualizovaná jako matice. Je možné v ní podobně jako v PerNeuronValue filtrovat hodnoty, stejně tak lze nakreslit histogram. Ten je stejný jako v 6.16.



Obrázek 6.18 Jak bylo vidět na obrázku 6.17, občas může být na hlavní úhlopříčce hodnota, která výrazně vybočuje z ostatních. Tuto hodnotu je možné vyfiltrovat a zlepšit barevné rozložení.



Obrázek 6.19 AnalogSignalList je zobrazená jako graf neuronů a graf seznamů pro vybrané z nich.



Obrázek 6.20 Čárový graf lze zvětšovat podle osy X. Při pohybu myší se zvýrazní nejbližší bod grafu, jeho lomená čára a dotyčný neuron.

Závěr

Navrhli jsme a implementovali webovou aplikaci pro vizualizaci modelu a datových struktur vygenerovaných za běhu Mozaiku. Splnili jsme všechny vytyčené požadavky.

Přenos velkého objemu dat ze serveru je zajištěný streamováním ve formátu CSV. Je snadné část dat přenášených po síti ve formátu JSON začít také streamovat ve formátu CSV bez změny klienta.

Specifikace složek s datastore je možná ve spouštěcím skriptu.

Přehledný výpis ADS byl dosažen ve formě stránkované tabulky s buňkami specializovanými na různé datové typy.

Filtrování a vyhledávání ADS je možné buď pomocí SQL, nebo pomocí dialogových oken.

Workspace s ADS je implementováno ve formě záložek.

Vícenásobné zobrazení je podporováno.

Sdílené nastavení ADS je rovněž podporováno.

Connections jsou vizualizovány jako interaktivní scatterplot.

PerNeuronValue je vizualizována jako obarvený interaktivní scatterplot.

PerNeuronPairValue je vizualizována jako interaktivní matice.

AnalogSignalList je vizualizován jako interaktivní kombinace scatterplotu a čárového grafu.

Udržovatelnost byla dosažena volbou frameworku, čistým a přehledným kódem a velkým množstvím automatických testů.

Možná rozšíření

Alternativní barevné škály

V různých případech se hodí použít různé barevné škály, v závislosti na tom, jaké informace jsou pro uživatele důležité a jak s daty chce zacházet[6]. Zatím neexistuje v aplikaci možnost vybírat z jiných škál než z periodické a neperiodické. Tato funkčnost by přinesla možnost efektivnější analýzy.

Permutace maticové vizualizace

Současná maticová vizualizace je užitečná pro rychlé zjištění specifické hodnoty. Je ale obtížné z ní vypozorovat existující trendy. Pro takovou funkčnost je nutné matici vhodně uspořádat[7]. Bylo by vhodné implementovat jeden nebo více existujících algoritmů, například eliptickou seriaci[8]

Seznam použitých zdrojů

[1] Computational Systems Neuroscience Group. *Mozaik documentation: Tutorial*. 2013. URL: https://github.com/CSNG-MFF/mozaik/blob/master/doc/source/tutorial.1.rst.

[2] Computational Systems Neuroscience Group. *Mozaik documentation: Introduction*. 2012. URL: https://github.com/CSNG-MFF/mozaik/blob/master/doc/source/introduction.rst.

[3] Kateřina Čížková. *NPRG035 Nástroj pro prohlížení modelů v systému Mozaik*. 2020. URL: https://github.com/antolikjan/mff_nprg045/blob/master/documentation.pdf.

[4] Mathias Wulff. *AlaSQL issue 1240: AlaSQL v3*. 2020. URL: https://github.com/AlaSQL/alasql/issues/1240.

[5] Computational Systems Neuroscience Group. *Mozaik README*. 2023. URL: https://github.com/CSNG-MFF/mozaik.

[6] Samuel Silva, Joaquim Madeira a Beatriz Sousa Santos. „There is more to color scales than meets the eye: a review on the use of color in visualization". In: *2007 11th International Conference Information Visualization (IV'07)*. IEEE. 2007, s. 943–950.

[7] Chun-houh Chen, Wolfgang Karl Härdle a Antony Unwin. *Handbook of data visualization*. Springer Science & Business Media, 2007.

[8] Chun-Houh Chen. „Generalized association plots: Information visualization via iteratively generated correlation matrices". In: *Statistica Sinica* (2002), s. 7–29.

Příloha A

Netextové přílohy

Součástí práce jsou přiložené zdrojové kódy a ukázkový datastore. Zdrojové kódy jsou k dispozici také v následujícím GitHub repozitáři.

`https://github.com/filipjezek/nprg045`

Příloha B

Dokumentace API

Tato dokumentace byla vygenerována z popisu API v souboru `/docs/openapi.json`

API Reference

Mozaik Visualization

API Version: 1.0.0

INDEX

API

1. ADS

Everything about analysis data structures

1.1 GET /analysis_ds

Get specific ADS

Returns different schema of ADS depending on the provided identifier. May require to issue other requests to get all of the data.

REQUEST

QUERY PARAMETERS

NAME	TYPE	DESCRIPTION
*path	string	
*identifier	enum ALLOWED: SingleValue, SingleValueList, PerNeuronValue, PerNeuronPairValue, AnalogSignal, AnalogSignalList, PerNeuronPairAnalogSignalList, ConductanceSignalList, Connections	
stimulus	object	
sheet	string	
*algorithm	string	
valueName	string	
neuron	string	
tags	array of string	

RESPONSE

STATUS CODE - 200: Successful operation

RESPONSE MODEL - application/json

```
{
  tags        [string]
  unit        string
  sheet       string
  neuron      integer
  period      number
  stimulus {
  }
  algorithm   string
  valueName   string
  identifier  enum      ALLOWED:SingleValue, SingleValueList, PerNeuronValue,
                        PerNeuronPairValue, AnalogSignal, AnalogSignalList,
                        PerNeuronPairAnalogSignalList, ConductanceSignalList,
                        Connections
}
```

STATUS CODE - 400: Invalid parameters or unimplemented ADS type

1.2 GET /analysis_ds/all

Get list of all ADS

REQUEST

QUERY PARAMETERS

NAME	TYPE	DESCRIPTION
*path	string	

RESPONSE

STATUS CODE - 200: Successful operation

RESPONSE MODEL - application/json

```
[{
Array of object:
  tags        [string]
  unit        string
  sheet       string
  neuron      integer
  period      number
  stimulus {
  }
  algorithm   string
  valueName   string
  identifier  enum      ALLOWED:SingleValue, SingleValueList, PerNeuronValue,
                        PerNeuronPairValue, AnalogSignal, AnalogSignalList,
                        PerNeuronPairAnalogSignalList, ConductanceSignalList,
                        Connections
}]
```

STATUS CODE - 400: Invalid parameters supplied

1.3 GET /analysis_ds/asl

Get ASL values

REQUEST

QUERY PARAMETERS

NAME	TYPE	DESCRIPTION
*path	string	
stimulus	object	
sheet	string	
*algorithm	string	
valueName	string	
neuron	string	

NAME	TYPE	DESCRIPTION
tags	array of string	

RESPONSE

STATUS CODE - 200: Successful operation

RESPONSE MODEL - text/csv

string

STATUS CODE - 400: Invalid parameters supplied

1.4 GET /analysis_ds/pnpv

Get PNPV values

REQUEST

QUERY PARAMETERS

NAME	TYPE	DESCRIPTION
*path	string	
stimulus	object	
sheet	string	
*algorithm	string	
valueName	string	
neuron	string	
tags	array of string	

RESPONSE

STATUS CODE - 200: Successful operation

RESPONSE MODEL - text/csv

string

STATUS CODE - 400: Invalid parameters supplied

2. FILESYSTEM

Access to the filesystem

2.1 GET /fs/directory

Get listing of a directory

Lists all datastores and directories in the provided directory

REQUEST

QUERY PARAMETERS

NAME	TYPE	DESCRIPTION
path	string	

RESPONSE

STATUS CODE - 200: Successful operation

RESPONSE MODEL - application/json

```
{
  name      string
  content   [string]
  datastore boolean
}
```

STATUS CODE - 400: Invalid path supplied

2.2 GET /fs/recursive

Get recursive listing of a directory

Lists all directories in the provided directory and its descendants. Collapses directory paths and filters out paths with no datastores.

REQUEST

QUERY PARAMETERS

NAME	TYPE	DESCRIPTION
path	string	

RESPONSE

STATUS CODE - 200: Successful operation

RESPONSE MODEL - application/json

```
{
  name      string
```

```
  content [{
  Array of object:
  }]
  datastore boolean
}
```

STATUS CODE - 400: Invalid path supplied

---

3. MODEL

Access to sheets and connections

3.1 GET /model

Get model metadata

Get sheet names and count of neurons for each of them, as well as count of connections between each of them

REQUEST

QUERY PARAMETERS

NAME	TYPE	DESCRIPTION
*path	string	

RESPONSE

STATUS CODE - 200: Successful operation

RESPONSE MODEL - application/json

```
{
  sheets [{
  Array of object:
    size   integer
    label  string
  }]
  connections [{
  Array of object:
    src    string
    size   integer
    target string
  }]
}
```

STATUS CODE - 400: Invalid parameters supplied

3.2 GET /model/positions

Get neuron positions for a given sheet

REQUEST

QUERY PARAMETERS

NAME	TYPE	DESCRIPTION
*path	string	
*sheet	string	

RESPONSE

STATUS CODE - 200: Successful operation

RESPONSE MODEL - text/csv

`string`

id,x,y

STATUS CODE - 400: Invalid parameters supplied

3.3 GET /model/connections

Get connections between two sheets

REQUEST

QUERY PARAMETERS

NAME	TYPE	DESCRIPTION
*path	string	
*src	string	
*tgt	string	

RESPONSE

STATUS CODE - 200: Successful operation

RESPONSE MODEL - text/csv

`string`

srcIndex,tgtIndex,weight,delay

STATUS CODE - 400: Invalid parameters supplied