Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Generátor podkresové hudby

Plzeň 2016 Zdeněk Janeček

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 12. května 2016

Zdeněk Janeček

Abstract

This thesis focuses on the possible techniques of modelling polyphony music using the combination of Recurrent Neural Network and Restricted Boltzmann Machine. Such a combination allows to distinguish the temporal dependencies in high-dimensional sequences along with high-level harmony learning.

The first chapter describes the task of music generation in a wider context and defines basic data representations. The following two chapters intruduce probabilistic models used for Polyphonic Music Generation in a general context. The second to last chapter deals with training the described models using gradient methods in the combination of Contrastive Divergence and Backpropagation Through Time. The last chapter of the thesis resolves the implementation and evaluation of the LSTM-RBM model.

Abstrakt

Tato práce popisuje možné techniky modelování vícehlasé hudby použitím kombinace rekurentních neuronových sítí a Omezených boltzmanových strojů. Tato kombinace umožnila rozpoznat temporální závislosti v mnohodimenzionálních sekvencích včetně harmonie.

První kapitola popisuje úlohu generování hudby v širších souvislostech, definuje základní pojmy a reprezentace dat. Další kapitoly představují pravděpodobnostní modely používané pro generování hudby v plné obecnosti. Předposlední kapitola se zabývá trénováním popsaných modelů pomocí gradientních metod v kombinaci s kontrastivní divergencí a zpětnou propagací v čase. Poslední část práce řeší implementaci a ohodnocení LSTM-RBM modelu.

Obsah

Seznam obrázků

1 Vymezení úlohy a základní pojmy

Umělecká činnost byla vždy doménou lidí. Pokrok umělé inteligence v posledních deseti letech prokázal, že jsme schopni dát počítačům komplexní rozhodování a napodobit tak lidskou inteligenci. To ale nestačí k tomu, aby stroj tvořivě myslel. Je třeba mu dodat to, co dělá z člověka umělce. Tento požadavek se těžko uchopuje. Jedna cesta vede přes učení z již hotových děl. Přiznejme si, že i někteří slavní umělci využívají motivů nejen od sebe, ale i od ostatních.

V této práci se zajímám konkrétně o tvorbu (generování) hudby. Jedná se o velmi rozsáhlé téma, které má zatím ke komplexnímu řešení daleko. Je třeba si tedy včas stanovit hranice, kam až zajít. Mezi taková omezení patří:

- generátor nebude primárně řešit celkový průběh skladby (v hudebním světě sloky, mezihry a tedy změny nebo opakování témat hudby),
- bude se generovat jeden nástroj,
- harmonie se může měnit, ale jen v rámci natrénovaného vzoru,
- změny tempa, ale délky not budou tempu odpovídat,
- hlasitost ani barva nástroje.

Výsledný model nebude potřebovat apriorní znalost harmonie. Hodnocení kvality generované hudby bude prováděno statisticky podle vzdálenosti generovaného a očekávaného výsledku.

1.1 Střípky z hudební teorie

Slovo *hudba* vzniklo od slovesa „hráti“ a označovalo hru na hudební nástroj. Na takový nástroj hrál hudebník, kterému se začalo lidově říkat muzikant. Cílem této práce tedy není nahrazení hudebníků, ale vytvoření rozšířené hudební tvorby za pomoci umělé inteligence, přestože primárním hudebníkem (syntezátorem) bude v našem případě také počítač. Jedná se tedy o jiný způsob skládání hudby.

Vědní obor, který zkoumá vše kolem hudby se jmenuje *Muzikologie* a jedná se z velké části o humanitní vědu. Důležitou exaktní součástí je právě hudební teorie, ve které nalezneme popis zákonitostí tónového uspořádání.

První pojmem je *délka not*. Za základní délku můžeme považovat celou notu. Další noty mohou být kratší podle racionálního zlomku. Máme tedy noty půlové, čtrťové a tak dále.

Každý tón má svojí výšku. Ta se dá přepočítat na ekvivalentní frekvenci zvuku. Má-li tento tón své jméno, pak se jedná o *notu*. Záleží na ladění těchto tónů, ale může se stát, že více not znamená jeden tón (Janeček, 2014), proto nebudeme hovořit o notách, ale o tónech.

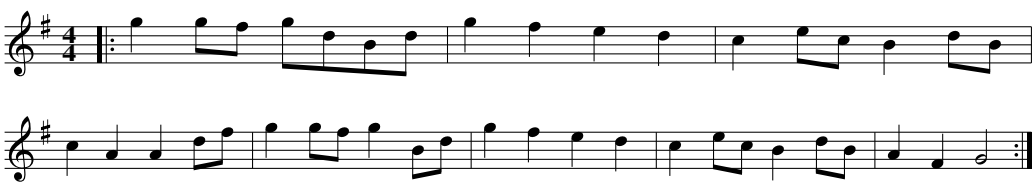
Zní-li více tónů dohromady, tvoří *vícehlasou hudbu*. Některé souzvuky jsou tak známé, že jsou pojmenované a tvoří *akordy*. Vzdálenost mezi současně znějícími tóny se nazývá *interval*. Ten se v běžné hudbě měří v půltónech.

1.2 Záznam hudby

1.2.1 Notová osnova

Nejznámnějším druhem záznamu hudby je *notová osnova* na obrázku 1.1. Její výhodou je snadné čtení, protože je založena na obrázkových symbolech. Hlavička noty určuje výšku a délka je určena tvarem. Všechny noty jsou vztažené relativně k referenčnímu tónu, který je pevně určen klíčem na začátku řádky.

Kombinací předznamenání (symboly hned za klíčem), které určuje posuny některých tónů a klíče, dostáváme tón v absolutní výšce.



Obrázek 1.1: Ukázka notové osnovy vysázené z ABC notace.

1.2.2 ABC

Oblíbeným zápisem pro jednohlasou folkovou hudbu se stal formát *ABC*. Vzniknul původně pro jednohlasou hudbu, ale od verze 2 je možné zapsat i vícehlas. Základním požadavkem je snadné čtení člověkem a přenositelnost pomocí internetu. Jedná se o ASCII soubor:

```
X:1
M:4/4
L:1/8
K:G treble
|:g2gf gdBd|g2f2 e2d2|c2ec B2dB|c2A2 A2df|
  g2gf g2Bd|g2f2 e2d2|c2ec B2dB|A2F2 G4:|
```

kde `X` určuje počet písní v souboru, `M` označení dob v taktu, `L` základní délka noty, `K` je tónina *G dur* a houslový klíč. Dále následuje melodie popsaná hudebním označením tónů. Vysázená notová osnova je na obrázku 1.1.

Zapsaná nota stále neoznačuje absolutní výšku tónu a pro její získání je třeba vzít v úvahu ještě tóninu. Tento formát se přesto stal populární pro generování hudby. Sturm et al. (2016) používají rozsáhlou síť LSTM modulů (viz kapitola 3.4) jako znakový nebo symbolický (více než jeden znak) model generující ABC formát.

1.2.3 Obrázková partitura

Tento formát se odlišuje od předchozích tím, že používá absolutní výšky tónů. Tóny jsou uspořádány do matice. V jednom rozměru je 88 tónů klaviatury a další rozměr je čas. Nejmenší dílek délky tónu lze volit podle aplikace. Převod do tohoto fomátu včetně ukázky popisuji v kapitole 5.2.

Tento formát záznamu se objevil poprvé v *hrací skříně*, *orchestrionu* nebo *flašinetu*. První zmínky pochází už z 16. století. Tehdejší záznam byl rozdílný v tom, že nebyl uložen jako dvojrozměrná matice, ale na válečku různých materiálů. Ukázka vnitřku flašinetu je na obrázku 1.2.

Obrázek 1.2: Ukázka historické verze obrázkové partitury ve formě válečku. Zdroj: Wikipedia

Díky výzkumu generování hudby v obrázkové partituře nás možná čeká renesance těchto hracích strojků, které poběží v našem chytrém telefonu, jen s tím rozdílem že váleček bude nekonečný a hudba stále originální.

1.3 Historie generované hudby

Tvorba hudby na počítači je spjata s jejich vývojem. Počítače nejdříve sloužily jen jako řídící zařízení mnoha analogových syntezátorů. Tím vznikla potřeba synchronizovat nekompatibilní zařízení od různých výrobců, a proto vznikl standard MIDI. Postupný vývoj číslicové techniky umožnil vytvářet a reprodukovat hudbu čistě digitálně. V polovidě osmdesátých let tak bylo možné programovat, co bude syntezátor hrát. Takovým přístrojem byla například řada syntezátorů *Fairlight CMI*. Na těchto přístrojích dokonce běžel operační systém *QDOS*. Hudba byla v té době ještě skládaná člověkem.

Skutečné generátory hudby vznikly až polovině devadesátých let. Prvním z nich byl *SSEYO Koan*[1], který byl představen v roce 1994 a jednalo se o tzv. Ambientní hudbu a tedy první generovanou podkresovou hudbu.

Nejdříve byla hudba jen živá, poté reprodukovaná a nyní máme hudbu generativní, která kombinuje výhody obou svých předků. Stejně jak je živá hudba pokaždé interpretována jinak, tak je možné ji poslouchat kdykoliv. To je velkou motivací této práce.

[1] `http://www.intermorphic.com/sseyo/koan/generativemusic1`

2 Dostupné metody

2.1 Problematika

V předchozí kapitole jsem popsal různé druhy záznamů hudby a jejich výhody. Při její tvorbě je třeba vzít v úvahu jak kontext prostorový, tak časový. Prostorovým kontextem myslím tóny hrající současně. Tóny by měly znít jen v omezené množině vzájemných intervalů, protože jiné intervaly nezní dobře.

Zvládnutí komplikovaných systémů, v mém případě hudby, vyžaduje dvě věci: držet model modulární a dostatečně abstraktní. To je známý mechanismus strojového učení. Naší snahou je najít model podmíněné pravděpodobnosti náhodné proměnné $p(\mathbf{x}|\theta)$. Z tohoto modelu potřebujeme efektivně odvodit následující řešení řetězce $x_{1:T}$ o délce T (Murphy, 2012). Naším cílem je modelovat vícedimensionální objekty v každém časovém kroku, podmíněné všemy předchozímy.

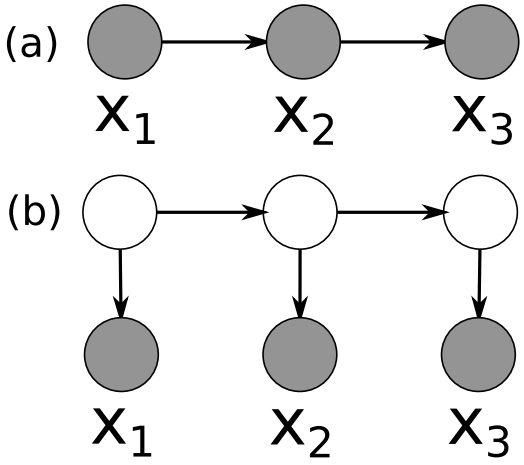
Při pozorování zkoumaného jevu zjistíme, že náhodné jevy jsou korelované s některým skrytým stavem. Model, který využívá tyto stavy, se také nazývá *Latent Variable Model* (LVM), protože popisuje latentní (nepozorované) závislosti náhodných proměnných. Tyto modely jsou hůře trénovatelné, ale potřebují méně parametrů. Často se využívají jako *úzké hrdlo* (angl. bottleneck), které vytváří „komprimovanou“ podobu dat. Základní modelem je směs (angl. Mixture). Model se skládá ze základních distribucí pravděpodobnosti, kdy jejich váženým součtem získáme výslednou pravděpodobnost daného vstupu. Tato váha může být definována jako *bránová funkce* (angl. gating function). Pak se jedná o *Směs expertů* (angl. Mixture of experts). Máme-li k dispozici model dat (například lineární regresi v rovnici 2.2), zjistíme, že není platný v celém rozsahu vstupů. Bránová funkce pak tedy určuje kdy je jaký model platný. To vychází z myšlenky, že každý podmodel je expertem číslo k jen v určitém regionu.

Pro následující definice je potřeba definovat normální rozdělení pravděpodobnosti:

$$\mathbb{N}(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{1}{2\sigma^2}(x-\mu)^2} \tag{2.1}$$

Pak můžeme definovat lineární regresi jako:

$$p(y|\boldsymbol{x}, z=k, \boldsymbol{\theta}) = \mathbb{N}(y|\mu_k(\boldsymbol{x}), \sigma_k^2), \text{ kde } \mu_k(\boldsymbol{x}) = \boldsymbol{W}_k^T\boldsymbol{x} + \boldsymbol{b}_k \tag{2.2}$$



Obrázek 2.1: (a) Markovský model prvního řádu (b) Skrytý markovský model HMM

Bránová funkce $p(z|\boldsymbol{x}, \boldsymbol{\theta})$ určí jakého experta použít. Výslednou predikci získáme:

$$p(y|\boldsymbol{x}, \boldsymbol{\theta}) = \sum_k p(z = k|\boldsymbol{x}, \boldsymbol{\theta}) p(y|\boldsymbol{x}, z = k, \boldsymbol{\theta}) \tag{2.3}$$

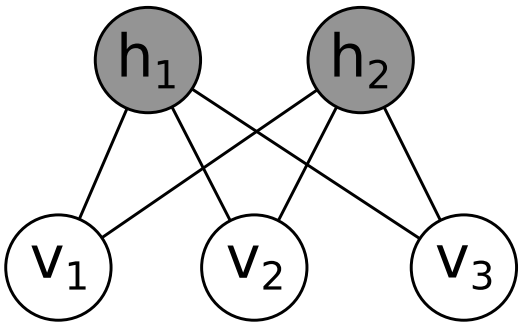
Na stejném principu je pak založen *Součin expertů* (angl. Product of experts) (Hinton, 2002). Tento model může určit různá omezení a dosáhnout jednoznačnějších výsledků. *Restricted Bolzman Machines* (RBM) popsané v kapitole 2.3.1 jsou právě speciálním případem tohoto modelu.

2.2 Markovské modely

Jedná se o matematický aparát k popisu náhodných procesů, které představují přechody z jednoho stavu do druhého. Této sekvenci náhodných proměnných se říká Markovský řetězec a modeluje se pomocí řetězového pravidla:

$$p(x_{1:T}) = p(x_1) \prod_{t=2}^{T} p(x_t|x_{t-1})$$

S výhodou lze využít předpokladu, že budoucí hodnoty řetězce jsou nezávislé na těch z minulosti. Tomu se říká Markovský model prvního řádu. Ten je charakterizován počáteční distribucí a přechodovou maticí ze stavu i do j pravděpodobností $p(x_t = j|x_{t-1} = i)$. Tento model je základem většiny dalších. Jedná se vlastně o rozšíření konečných automatů, kde je pravděpodobnost přechodu určena váhou hrany. Některé přechody nejsou popsa-



Obrázek 2.2: RBM se třemi uzly ve viditelné a dvěma uzly ve skryté vrstvě.

telné v reálném světě přímo. Proto vzniklo rozšíření *Skrytý Markovský model* (angl. Hidden Markov model). Oba modely jsou na obrázku 2.1. Stavem tohoto řetězce může být nějaká abstraktní vlastnost, například téma slova jako „biologie“. Tento stav jen ovlivní výstup. Kombinací nepřímých důkazů lze složit komplikovanější závěr.

Tento model se trénuje například pomocí *Viterbiho algoritmu*, *EM*, *Dopředná procedura*, nebo *Baum-Welcha*. (Jurafsky – Martin, 2014) (Brychcín, 2010)

2.3 Grafické modely

Zobecněním Markovských řetězců vznikly grafické modely. Díky nim lze popisovat komplikovanější děje například s obrázky, nebo 3D objekty. Uzly jsou náhodné proměnné libovolného rozměru a hrany jsou přechody mezi nimy. Tyto hrany mohou být orientované i neorientované stejně jako v grafu.

2.3.1 Neorientované grafy

Tyto modely neurčují směr přechodu. Jejich vazba je často stochastická a tedy výpočty jsou více výpočetně náročné. Výsledné parametry nejsou jednoduše interpretovatelné. Vrcholy mohou tvořit různé skupiny, které dohromady tvoří *Boltzmannův stroj*. Vrcholy v této skupině mohou mít také vnitřní spojení. Těmto modelům se také říká grafické nebo také Markovská náhodná pole (angl. Markov random fields – MRF). Spočítat likelihood (pradĕpodobností rozdělení dat) takového modelu je velmi výpočetně náročné a proto se používá technika *Markov Chain Monte Carlo* (MCMC), kterou podrobně popisuje (Fischer – Igel, 2012).

Jestliže omezíme *Boltzmannův stroj*, aby neměl žádné vnitřní spojení, zavedeme R viditelných uzlů a K skrytých a vytvoříme mezi viditelnými a

skrytými uzly bipartitní graf, získáme RBM. Každý skrytý uzel je nezávislý na ostatních, stejně jako v neuronové síti v kapitole 2.4. Ukázka jednoduchého RBM je na obrázku 2.2. Každý skrytý uzel reprezentuje určitou vlastnost vstupu. Na váhy, směřující do skrytého uzlu, lze pohlížet jako filtr vstupu $\boldsymbol{v}$. Zvyšováním počtu skrytých uzlů lze rozšířit jeho kapacitu. To ale navyšuje riziko přeučení, kde je většina naučených filtrů vysoce korelovaných. Z toho plyne vyšší náročnost modelu. Proto se snažíme držet velikost co nejmenší a získat co nejvíc rozdílných řešení. V kapitole 4 popisuji často používané metody optimalizace, které se využívají právě na trénování RBM.

RBM má mnoho variant podle toho, jaké rozdělení mají aktivace uzlů. Nejčastější je *binární RBM*. Pro něj platí, že skryté i viditelné uzly jsou binární. Sdružená pravděpodobnost:

$$p(\boldsymbol{v}, \boldsymbol{h}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \exp(-E(\boldsymbol{v}, \boldsymbol{h}; \boldsymbol{\theta})). \tag{2.4}$$

Vektor $\boldsymbol{v}$ je viditelná vrstva, $\boldsymbol{h}$ je skrytá a funkce E je energie systému:

$$E(\boldsymbol{v}, \boldsymbol{h}; \boldsymbol{\theta}) \triangleq -(\boldsymbol{v}^T \boldsymbol{W} \boldsymbol{h} + \boldsymbol{v}^T \boldsymbol{b} + \boldsymbol{h}^T \boldsymbol{c}). \tag{2.5}$$

V této rovnosti se nachází navíc matice vah $\boldsymbol{W}$ a příslušný bias viditelné a skryté složky $\boldsymbol{b}$ a $\boldsymbol{c}$.

Pro úplnost uvádím dělící funkci:

$$Z(\boldsymbol{\theta}) = \sum_{\boldsymbol{v}} \sum_{\boldsymbol{h}} \exp(-E(\boldsymbol{v}, \boldsymbol{h}; \boldsymbol{\theta})). \tag{2.6}$$

Její výpočet by zabral v nejhorším případě $\mathcal{O}(2^R 2^K)$ času a proto je přímý výpočet nespočitatelný. Parametry systému jsou $\boldsymbol{\theta} = (\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c})$.

Pro posterior rozdělení pravděpodobnosti aktivace RBM uzlů platí:

$$p(\boldsymbol{h}|\boldsymbol{v}, \boldsymbol{\theta}) = \prod_k \mathrm{Ber}(h_k|\sigma(\boldsymbol{W}_{:,k}\boldsymbol{v} + \boldsymbol{c})) \tag{2.7}$$

$$p(\boldsymbol{v}|\boldsymbol{h}, \boldsymbol{\theta}) = \prod_r \mathrm{Ber}(v_r|\sigma(\boldsymbol{W}_{r,:}\boldsymbol{h} + \boldsymbol{b})) \tag{2.8}$$

Tato rovnice odpovídá právě zmiňovanému součinu expertů zmíněnému v části 2.1. Bernoulliho rozdělení je speciální případ binomického a je definováno jako:

$$\mathrm{Ber}(x|\theta) = \theta^{\mathbf{1}(x=1)}(1-\theta)^{\mathbf{1}(x=0)}.$$

Symbol $\mathbf{1}$ znamená charakteristickou funkci:

$$\mathbf{1}(e) = \begin{cases} 1 & e \text{ je pravda,} \\ 0 & e \text{ je nepravda.} \end{cases}$$

Oblíbenou funkcí ve strojovém učení je sigmoida, která mapuje nelineárně reálná čísla na interval $\langle 0, 1 \rangle$:

$$\sigma(x) \triangleq \frac{1}{1 + \exp(-x)}. \tag{2.9}$$

Rovnice 2.7 tedy znamená ve vektorovém počtu prostý součin matice a vektoru. Vidíme také, že k aktivaci uzlu k dojde podle toho, jak je vstupní vektor $\boldsymbol{v}$ „podobný“ sloupci matice $\boldsymbol{W}_{:,k}$.

V maticovém zápisu je výpočet pravděpodobnosti aktivace:

$$p(\boldsymbol{h}|\boldsymbol{v}, \boldsymbol{\theta}) = \sigma(\boldsymbol{W}^T \boldsymbol{v} + \boldsymbol{c}), \tag{2.10}$$

$$p(\boldsymbol{v}|\boldsymbol{h}, \boldsymbol{\theta}) = \sigma(\boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}). \tag{2.11}$$

V této neorientované variantě jsou váhy pro obě rovnice stejné, ale v určitém případě mohou být různé. Pak se nazývají *generativní* ty, které vedou k viditelnému vektoru a *rozpoznávací*, které vedou ke skrytému vektoru, protože jsou používány k rozpoznávání vstupu (například klasifikátorem nebo dalším RBM).

Pro získání skutečných aktivací je potřeba provést vzorkování z tohoto rozdělení. K tomu je potřeba generátor čísel rovnoměrného rozdělení. To jsem napsal pomocí Torch:

```
1   p:csub(torch.rand(p:size())):sign():clamp(0, 1)
```

Důležitější než sdružená energie celého RBM v rovnici 2.5 je volná energie vstupu:

$$\mathcal{F}(\boldsymbol{v}) = -\boldsymbol{v}^T \boldsymbol{b}_v - \sum_i \log(1 + \exp(\boldsymbol{W}\boldsymbol{v} + \boldsymbol{b}_h))_i \tag{2.12}$$

V této práci jsem jiný model nepoužil, ale jen pro úplnost připomenu, že lze také použít jak Gaussovské vstupy, tak skryté stavy. Vstupy mohou mít více kategorií nebo témat. Ve spojitém světě se tak jedná o variantu *analýzy hlavních komponent* (PCA), a tedy redukci dimensionality podle (největšího) rozptylu dat.

Metody učení vychází z gradientních metod. K trénování se používá známá metoda *Contrastive Divergence* (více v kapitole 4.2). RBM lze snadno rozšířit do více vrstev, trénovat hladově a vytvořit tak *Deep Belief Network* (viz kapitola 2.4).

2.3.2 Orientované grafy

Tyto modely jsou známé spíše jako *Bayesovské sítě*. Přestože mají v názvu Bayes, nemají nic společného s Bayesovskou větou. Přesnější pojmenování je *Belief Network*, protože vyjadřuje důvěru nějakého vztahu mezi uzly.

Orientované grafické modely mají velmi blízko k RBM, které je rozebráno v následující kapitole. Velmi zajímavou alternativou do budoucna mohou tak být *Sigmoid Belief Network* (SBN) a zvlášť jejich časová varianta TSBN (Gan et al., 2015). Při řetězení SBN vzniká závislost na předchozích časových krocích.

SBN modeluje také vztah mezi viditelnými $\boldsymbol{v} \in \{0,1\}^M$ a skrytými uzly $\boldsymbol{h} \in \{0,1\}^J$, propojené vahami $\boldsymbol{W} \in R^{M \times J}$. Přechod mezi skrytým a viditelným uzlem je opět:

$$p(v_m = 1|\boldsymbol{h}) = \sigma(\boldsymbol{W}_{:,m}^T \boldsymbol{h} + b_m), \qquad p(h_j = 1) = \sigma(b_j).$$

Struktura je opět bipartitní graf. Energie může být zapsána jako:

$$-E(\boldsymbol{v}, \boldsymbol{h}; \boldsymbol{\theta}) \triangleq \boldsymbol{v}^T \boldsymbol{W} \boldsymbol{h} + \boldsymbol{v}^T \boldsymbol{b} + \boldsymbol{h}^T \boldsymbol{c} - \sum_m \log(1 + \exp(\boldsymbol{W}_{:,m}^T \boldsymbol{h} + b_m)) \quad (2.13)$$

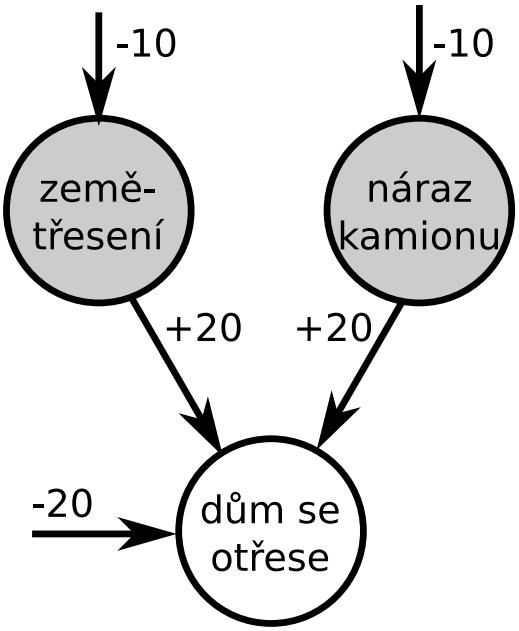
SBM tvoří přirozeně generativní proces. Skryté stavy poskytují přímé vysvětlení vstupu, ale jejich vliv je ovlivněn „efektem vyrušení“, popsaným v následující kapitole. Je třeba použít pokročilé metody odvozování, které navrhuje například Mnih – Gregor (2014).

Velmi oblíbenou variantou Belief net je také *Neural Autoregresive Distribution Estimator* – NADE, kterou navrhli Larochelle – Murray (2011). Tento model je inspirován RBM a jeho hlavním cílem je dosáhnout spočitatelné dělící funkce 2.6. Specializuje se na modely vysokodimenzionálních sdružených pravděpodobností. Jedná se o *Fully Visible Sigmoid Belief Network* – FVSBN. Podmíněná pravděpodobnost uzlu v_i je vyjádřena jako nelineární funkce proměnné v_k pro $\forall k < i$. Rozdíl mezi RBM a NADE je v gradientu negativního likelihood, který není již odhadován pomocí *Contrastive Divergence* (viz kapitola 4.2) jako v RBM. Díky tomu je možné použít lepší nelineární optimalizační metody.

Podmíněná pravděpodobnost $p(v_i = 1|\boldsymbol{v_{<i}})$ se dá přepsat jako:

$$p(v_i = 1|\boldsymbol{v}_{<i}) = \sigma(\boldsymbol{W}_{:,i}^T \boldsymbol{h} + b_i) \quad \boldsymbol{h}_i = \sigma(\boldsymbol{W}_{<i,:} \boldsymbol{v}_{<i} + \boldsymbol{c})$$

Každé $p(v_i = 1|\boldsymbol{v_{<i}})$ odpovídá neuronové síti s jednou skrytou vrstvou a svázanými vahami do a ven ze skryté vrstvy. To značně urychluje výpočet. Bez této optimalizace je složitost výpočtu $p(\boldsymbol{v})$ $\mathcal{O}(HD)$. Kde H je počet skrytých neuronů a D vstupních.



Obrázek 2.3: Jednoduchá Belief síť obsahující dvě nezávislé příčiny, že se dům otřese.

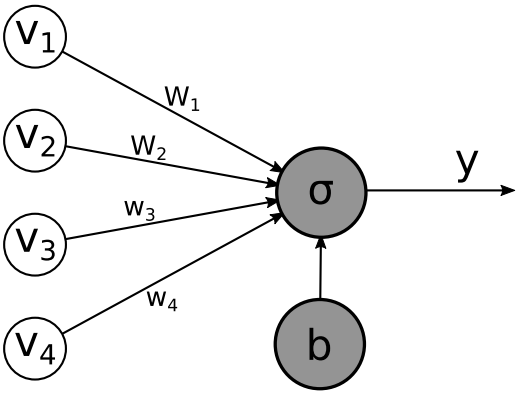
2.3.3 Vzájemné vyrušení

Orientované grafické modely se potýkají s velmi nepříjemným jevem vzájemným vyrušením faktů (angl. Explaining Away). Efekt skrytých uzlů na viditelné může být silně korelovaný a to dělá odvozování ze sdružené pravděpodobnosti modelu problematické (Hinton et al., 2006).

Tento efekt je ilustrován na obrázku 2.3. Tyto dvě příčiny se mohou stát snadno antikorelované, protože mají stejný bias. Ten je záporný, takže uzly příčin budou e^{10} krát méně pravděpodobně aktivní. Při aktivaci jedné příčiny bude celková aktivace uzlu, že dům se otřese, nulová. To mu dává šanci být aktivní a je to také rozumnější vysvětlení, proč se dům otřásl, než žádný vstup kdy bude mít aktivace hodnotu svého biasu e^{-20}.

Jestliže se ale aktivují obě příčiny, pak vzniklá pravděpodobnost otřesu domu je $e^{-10} \times e^{-10} = e^{-20}$. Díky tomu se obě příčiny vzájemně vyruší.

Tento jev se řeší částečným odstraněním orientace hran, a to tzv. spojením vah. Toho je v využíváno v kapitole 2.5 pro trénování sítí s hlubokým učením.



Obrázek 2.4: Ilustrace modelu neuronu.

2.4 Neuronová síť

V této kapitole bych rád rychle připomněl, co neuronová síť je, protože jsem se o ní několikrát zmínil. Neuronové sítě jsou elegantní biologicky inspirovaná paradigmata, která se umí učit z dat.

Základní neuronovou sítí je *Vícevrstvý perceptron* (angl. Multilayer Perceptron). Základním modelem neuronu je *McCulloch-Pittsův neuron* (obrázek 2.4), který je definován:

$$\boldsymbol{y}(\boldsymbol{v}) = \mathcal{S}(\boldsymbol{W}\boldsymbol{v} + \boldsymbol{b}). \tag{2.14}$$

Funkce $\mathcal{S}$ může být libovolná nelineární aktivační funkce. Základní funkcí je Sigmoida z rovnice 2.9. Ukazuje se, že tako aktivační funkce má problémy s mizením gradientu (angl. vanishing gradients), a proto se postupně prosazují *Rectifier Linear Units* – ReLU (Nair – Hinton, 2010).

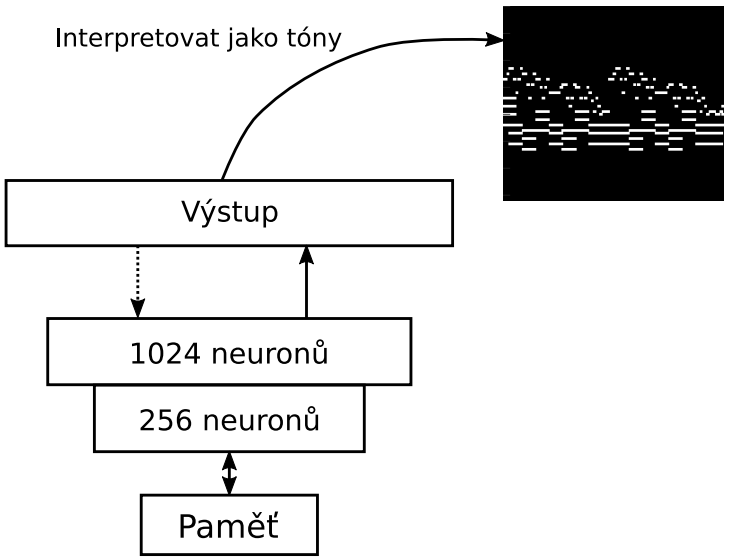
Neuron provede lineární projekci vstupu $\boldsymbol{v}$ maticí vah $\boldsymbol{W}$ a pak nelineární transformaci aktivační funkcí.

Neuronová síť bez skrytých vrstev je schopná naučit se jen lineárně separativní problémy. Přidáním dostatečného počtu skrytých vrstev dokáže modelovat kteroukoli hladkou funkci.

Základním trénovacím algoritmem je *Perceptron*. Jeho parametrem je práh t. Nebo je možné použít bias, ale pak je práh nulový. Aktivace menší než práh neuron neaktivují:

$$\hat{y}(\boldsymbol{v}) = \begin{cases} 1 & \boldsymbol{w}^T\boldsymbol{v} \geq t, \\ 0 & \boldsymbol{w}^T\boldsymbol{v} < t. \end{cases}$$

Vektor vah $\boldsymbol{w}$ je velký jako vektor vstupů $\boldsymbol{v}$. Neuron ve své podstatě představuje binární logistickou regresi. Při klasifikaci přiřazujeme vstupu $\boldsymbol{v}$



Obrázek 2.5: Použití natrénovaného dekodéru ke generování partitury.

třídu y_i. Výsledný gradient podle vah je $(\hat{y}_i - y_i)\boldsymbol{v}$. S tímto gradientem se ještě mnohokrát setkáme. Jedná se totiž o gradient funkce cross-entropy:

$$l = -y_i \log \hat{y}_i - (1 - y_i) \log(1 - \hat{y}_i) \tag{2.15}$$

Výstup funkce 2.14 může vést na vstup dalšímu neuronu. Ty jsou tak uspořádány po vrstvách. Obecně může být vazba také rekurentní jako v kapitole 3.1. Neuronová síť tak má svojí vstupní, skrytou a výstupní vrstvu. Skrytých vrstev může být i více a důležité je, že jsou vrstvy úplně propojené. V knihovně Torch7[1] se jedná o modul `nn.Linear`. Jejich propojení je zajištěno pomocí kontejnerů.

Podobnost s předchozími modely není náhodná. Zásadní rozdíl je v tom, že Boltzmanovy stroje používají stochastické vazby. Trénování grafických modelů je díky tomu rozdílné. Používají se často k předtrénování parametrů neuronové sítě. Díky tomu dosahuje výsledná síť často lepších výsledků.

Jeden z prvních návrhů mé diplomové práce byl založen na *konvoluční neuronové síti* (angl. Convolutional Neural Network). Ta obsahuje zvláštní vrstvy sloužící jako banka filtrů. Díky architektuře recepčních polí získávají vlastnost prostorové nezávislosti, ale protože je generování z takového modelu náročné a nespolehlivé, tak jsem od něj upustil. Není přípustné, aby vznikal kolem tónu efekt překmitů, který vzniká na ostrých hranách. Tuto myšlenku rozvinul Johnson (2015) s využitím mnoha LSTM modulů (viz kapitola 3.4). Nevýhodou této sítě je mimo velkého počtu LSTM hlavně

[1] `http://torch.ch`

komplexnost vstupů. Zajímavá byla ve svém časově/prostorovém zapojení. Tóny tedy závisely jak na své historii, tak na historii jejich okolí.

Cílem optimalizace cenové funkce neuronové sítě je její minimalizace podle parciálních derivací vzhledem k vahám a biasu. Velmi dobrý rozbor k tomuto napsali LeCun et al. (1998). Jednotlivé kroky jsou následující:

- provedeme dopřednou propagaci zdola nahoru (získáme aktivace),
- spočítáme chybu výstupu $\hat{\boldsymbol{y}}$ vzhledem k vstupu: $\delta_n^L = \hat{y}_n - y_n$ (například pomocí střední kvadratické odchylky — $\frac{1}{N}\sum_n(\hat{y}_n - y_n)^2$ pro regresi nebo cross-entropy v rovnici 2.15 pro klasifikaci),
- zpětně propagujeme chybu do vrstvy l: $\boldsymbol{\delta}^l = (\boldsymbol{W}^{l+1})^T\boldsymbol{\delta}^{l+1}$,
- spočítáme celkový gradient cenové funkce příchozí chyby v každé vrstvě vzhledem k vstupu,
- aplikujeme optimalizační funkci na parametry modelu s použitím spočítaných gradientů (více v kapitole 4).

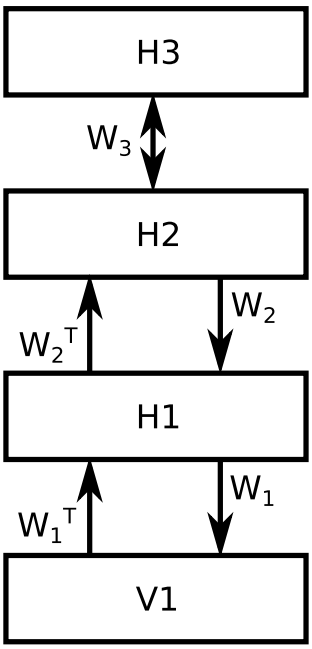
Na celý proces se doporučuje použít tzv. mini–batch (dávka). Na vstup tedy nejde jeden vektor, ale matice o velikosti $B \times V$. Přičemž B je velikost dávky a V vstupu. Některé operace se tedy v důsledku počítají v obráceném pořadí, než jak je uvedeno v teoretickém vztahu jako například $\boldsymbol{W}\boldsymbol{v}$. Matice vah je obvykle velikosti $H \times V$. Kdyby byl vstup zmíněného rozměru, nešlo by jej vynásobit. Proto se v dávkách násobí $\boldsymbol{V}\boldsymbol{W}^T$.

2.5 Sítě s hlubokým učením

Jsou v poslední době významnou technikou, která získala velkou popularitu kolem roku 2006 (Nielsen, 2015). Jedná se o jistou kombinaci orientovaných a neorientovaných modelů (obrázek 2.6). Dalším znakem je také velký počet skrytých vrstev a generativnost modelu.

Trénováním těchto modelů (angl. *Deep Belief Network* – DBN) se zabývají Hinton et al. (2006). Na obrázku 2.6 je ukázka hluboké sítě. Vrchní vrstvy tvoří neorientovanou stochastickou vazbu a asociativní paměť. Další vrstvy obsahují generativní a rozpoznávací orientované vazby. Při hladovém předtrénování jsou tyto vazby spojené a trénují se jako RBM.

Další aplikací DBN jsou tzv. *autoenkodéry*. Ty jsou stavěny tak, aby vytvářely úzké hrdlo (angl. bottleneck) a tím vynutily snižovat dimensionalitu dat. Síť se trénuje tak, že se rozdělí na *enkodér* a *dekodér*, který sdílí váhy a



Obrázek 2.6: Hybridní síť. Základ Deep Belief Network.

biasy. Enkodér vytváří skrytou reprezentaci (paměť) a dekodér interpretuje, co je v ní uloženo. Vytváří tak rekonstrukci zakódovaného stavu.

Trénují se jako *Deep Bolzmann Machines* (zkr. DBM), a to hladově po vrstvách. Výsledný dekodér lze pak použít jako generátor partitury o pevné velikosti jako na obrázku 2.5. Problémem takového návrhu je navazování těchto úseků. Další problém tohoto návrhu je velmi omezená kreativita (Felix, 2015).

3 Temporální modely

3.1 RNN

Nyní se budu věnovat specifickému druhu neuronové sítě. Dopředné sítě zaznamenaly v poslední době obrovský úspěch v mnoha aplikacích. Jejich nedostatkem je však omezenost zachytit závislosti v čase příchozí sekvence. Rekurentní síť může modelovat teoreticky nekonečné sekvence. Tato síť má také paměť. Díky tomu je schopná například predikovat další slovo. Také mají koncept skrytého stavu, ale ten je nyní schopen zachovat, co se dělo několik kroků zpátky.

Jedním ze známých problémů jsou *mizející gradienty*, které popsali již Bengio et al. (1994). Ty se projevují typicky při použití naivního gradientního sestupu v hluboké síti. Rekurentní síť je automaticky hluboká se sdílením vah při jejím rozbalení v čase. Kvůli problémovému trénování tedy vzniká kolize se zachycením dlouhodobých závislostí. To je přesně proti našim požadavkům.

Vzhledem k tomu, že z podstaty věci je RNN hluboká, jeví se pojem „hluboká RNN“ nejednoznačný (Pascanu et al., 2013). V této práci jsem se zabýval jen mělkou rekurencí, ale ukázalo se, že by hlubší rekurence v čase hodně pomohla, a to mnohem více než DBN místo RBM.

RNN tedy simuluje dynamický systém v diskrétním čase. Na vstup $\boldsymbol{x}_t$ získáme výstup $\boldsymbol{o}_t$. Skrytý stav ale závisí také na jeho předchozím stavu. To se zapisuje:

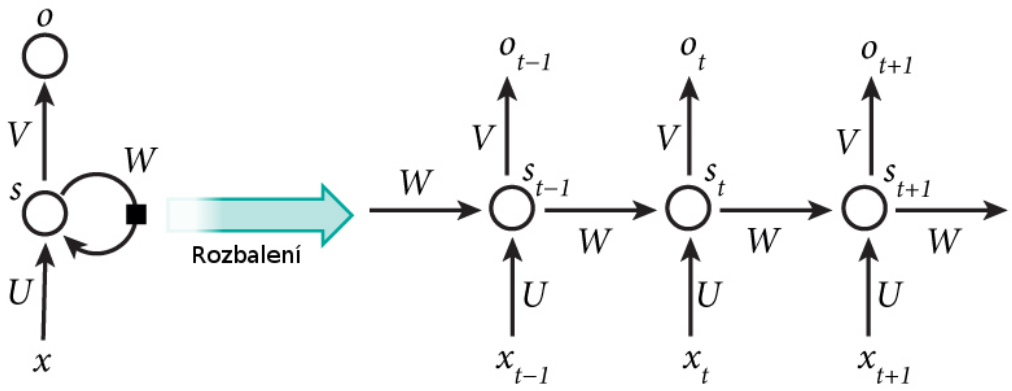
$$\boldsymbol{s}_t = f_h(\boldsymbol{x}_t, \boldsymbol{s}_{t-1}) \tag{3.1}$$

$$\boldsymbol{o}_t = f_o(\boldsymbol{s}_t) \tag{3.2}$$

Kde f_h a f_o jsou nelineární přechodové funkce. Minimalizací chybové funkce mezi očekávaným výstupem v konkrétním čase a samotným výstupem funkce f_o. Chybová funkce vypadá obecně:

$$E(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T} d\left(y_n^{(t)}, f_o(s_n^{(t)})\right) \tag{3.3}$$

kde vstupní vektor $\boldsymbol{x}$ je délky N, počáteční stav $\boldsymbol{s}^{(0)} = \mathbf{0}$ a $\boldsymbol{y}^{(t)}$ je požadovaný výstup. Funkce d nahrazuje příslušnou funkci chyby jako střední



Obrázek 3.1: RNN rozbalená v čase

kvadratická odchylka nebo třeba cross-entropy v rovnici 2.15. Délka sekvence je T.
Základním modelem je pak:

$$\boldsymbol{s}^{(t)} = \sigma(\boldsymbol{W}\boldsymbol{s}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}), \tag{3.4}$$

$$\boldsymbol{o}^{(t)} = \sigma(\boldsymbol{V}\boldsymbol{s}^{(t)}), \tag{3.5}$$

kde jednotlivé matice $\boldsymbol{W}$, $\boldsymbol{U}$ tvoří přechody skrytého stavu a $\boldsymbol{V}$ výstup. Takovou síť trénujeme pomocí *zpětné propagace v čase* (viz následující kapitola). Rozbalený model v čase je na obrázku 3.1. Tato síť je mělká z toho hlediska, že její přechody jsou jen výsledkem lineární projekce přechodů. Aby mohla být skutečně hluboká, musely by být tyto přechody podmíněné další rekurencí.

3.1.1 BPTT

Zpětná propagace v čase (Backprop through time — BPTT) je podobná té běžné. Počítá s tím, že se parametry v čase sdílí. Pro výpočet gradientu v čase $t = 5$ je třeba propagovat gradient zpět v čase. Všechny předchozí gradienty se sčítají.

Naším cílem je opět spočítat gradient chybové funkce vzhledem k parametrům $\boldsymbol{U}$, $\boldsymbol{V}$ a $\boldsymbol{W}$. Pro časové kroky platí:

$$\frac{\partial E}{\partial \boldsymbol{\theta}} = \sum_t \frac{\partial E^{(t)}}{\partial \boldsymbol{\theta}} \tag{3.6}$$

K výpočtu se používá řetízkové pravidlo. Například pro základní cross-entropy $E^{(t)} = -\boldsymbol{y}^{(t)} \log \boldsymbol{o}^{(t)} - (1 - \boldsymbol{y}^{(t)}) \log(1 - \boldsymbol{o}^{(t)})$, kde $\boldsymbol{o}^{(t)}$ je sigmoida

stavu $\boldsymbol{z}^{(t)}$, jejíž derivace je $\boldsymbol{o}^{(t)}(1-\boldsymbol{o}^{(t)})$ platí:

$$-\frac{\partial E^{(t)}}{\partial \boldsymbol{V}} = \frac{\partial E^{(t)}}{\partial \boldsymbol{o}^{(t)}} \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{z}^{(t)}} \frac{\partial \boldsymbol{z}^{(t)}}{\partial \boldsymbol{V}} = (\boldsymbol{o}^{(t)} - \boldsymbol{y}^{(t)}) \otimes \boldsymbol{s}^{(t)}$$

$$\boldsymbol{z}^{(t)} = \boldsymbol{V}\boldsymbol{s}^{(t)}$$

Derivace $\boldsymbol{V}$ závisí jen na aktuálních hodnotách v čase t. Operátor $\otimes$ znamená vnější součin vektorů, při kterém vzniká matice. Složitější je to pro ty ostatní parametry $\boldsymbol{W}$ a $\boldsymbol{U}$. Následující rovnost je platná:

$$-\frac{\partial E^{(t)}}{\partial \boldsymbol{W}} = \frac{\partial E^{(t)}}{\partial \boldsymbol{o}^{(t)}} \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{z}^{(t)}} \frac{\partial \boldsymbol{z}^{(t)}}{\partial \boldsymbol{W}}$$

Jenomže funkce 3.4 závisí na předchozích stavech. A tady nastupuje zpětný průchod. Jedná se konkrétně o $\boldsymbol{s}^{(t-1)}$. Musíme tedy nejdříve spočítat $\boldsymbol{s}^{(0)}$.

$$-\frac{\partial E^{(t)}}{\partial \boldsymbol{W}} = \sum_{k=0}^{t} \frac{\partial E^{(t)}}{\partial \boldsymbol{o}^{(t)}} \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{s}^{(t)}} \frac{\partial \boldsymbol{s}^{(t)}}{\partial \boldsymbol{s}^{(k)}} \frac{\partial \boldsymbol{s}^{(k)}}{\partial \boldsymbol{W}} \tag{3.7}$$

Díky tomu získáme příspěvek gradientu v čase (t) z každého časového kroku před ním. Počet kroků k je často omezen konstantou ρ. Tento algoritmus je již implementován v knihovně `rnn`. Tato redukce se spustí při zavolání `updateParameters(eta)`, nebo při použití kontejneru `nn.Sequencer`.

3.1.2 Mizející gradienty

Nyní se podívejme na problém mizejících gradientů (angl. Vanishing gradients) z pohledu nedávno definované derivace 3.7. Při pohledu na ní zjistíme že platí:

$$\frac{\partial \boldsymbol{s}^{(t)}}{\partial \boldsymbol{s}^{(k)}} = \prod_{j=k+1}^{t} \frac{\partial \boldsymbol{s}^{(j)}}{\partial \boldsymbol{s}^{(j-1)}}$$

Jedná se o Jakobián, který obsahuje derivace. Tyto hodnoty jsou na intervalu $\langle 0, 1\rangle$. Když je pak takto násobíme, klesá jejich hodnota exponenciálně a to způsobuje vymizení gradientů. Pak tedy nefungují ani dlouhodobé závislosti v čase. Stejný problém nastane pokud gradienty explodují až na NaN. Řešením je regularizace, ale preferované jsou v dnešní době ReLU aktivace místo sigmoidy či tanh. V kapitolách 3.4 a 3.5 navrhuji lepší RNN moduly, které řeší tento problém už svým návrhem. Prakticky neexistuje reálný funkční příklad, který by použil základní RNN a fungoval dobře.

3.2 RTRBM

Jedná se o první použitelnou temporální variantu RBM, kterou navrhli Sutskever et al. (2009), kde odlišně od *Temporal RBM* – TRBM lze dělat snadno odvozování. Zásadní rozdíl je v tom, že posloupnost skrytých stavů je oddělená od neorientovaného RBM a je v reálných číslech (říká se tomu *mean-field value* $\hat{h}^{(t)}$). To umožnilo velmi jednoduché odvozování.

Vychází z RBM, ale jeho parametry $b_v^{(t)}$, $b_h^{(t)}$ a $W^{(t)}$ jsou časově závislé. Jeho biasy pak tedy vznikají:

$$\boldsymbol{b}_h^{(t)} = \boldsymbol{b}_h + \boldsymbol{W}'\hat{\boldsymbol{h}}^{(t-1)}$$

$$\boldsymbol{b}_v^{(t)} = \boldsymbol{b}_v + \boldsymbol{W}''\hat{\boldsymbol{h}}^{(t-1)}$$

Během inference jsou skryté uzly $\boldsymbol{h}^{(t)}$ binární, ale při samplování nového času jsou to mean-field hodnoty a tedy:

$$\hat{\boldsymbol{h}}^{(t)} = \sigma(\boldsymbol{W}\boldsymbol{v}^{(t)} + \boldsymbol{b}_h^{(t)}) = \sigma(\boldsymbol{W}\boldsymbol{v}^{(t)} + \boldsymbol{W}'\hat{\boldsymbol{h}}^{(t-1)} + \boldsymbol{b}_h)$$

Tato rovnice je také podobná RNN přechodu stavu $\boldsymbol{s}^{(t)}$ v rovnici 3.4.

3.3 RNN-RBM

V kapitole 3.1 jsem popsal základní RNN. To slibuje schopnost uchovávat dlouhodobé závislosti, ale není snadné jej natrénovat. Dále jsem představil rekurentní variantu RTRBM. Boulanger-Lewandowski et al. (2012) navrhují jeho generalizaci. Tento model se stal základem dalších rozšíření jako RNN-DBN (Goel et al., 2014), DBN-BLSTM (Goel – Vohra, 2014), LSTM-RTRBM (Lyu et al., 2015) a také variantě LSTM-RBM, kterou jsem použil.

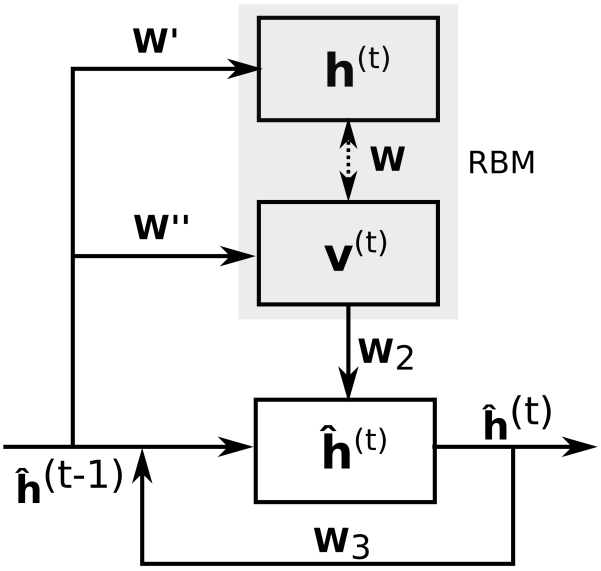
RTRBM lze chápat jako sekvenci podmíněných RBM, jejichž parametry jsou výstupem deterministické RNN, kde skryté uzly popisují temporální informaci. To nás vede k myšlence, že o temporální informace by se mělo starat jen RNN. Tento model je na obrázku 3.2.

RNN skryté stavy $\hat{\boldsymbol{h}}^{(t)}$ závisí jen na předchozím stavu a vstupu $\boldsymbol{v}^{(t)}$:

$$\hat{\boldsymbol{h}}^{(t)} = \sigma(\boldsymbol{W}_2\boldsymbol{v}^{(t)} + \boldsymbol{W}_3\hat{\boldsymbol{h}}^{(t-1)} + \boldsymbol{b}_{\hat{h}}) \tag{3.8}$$

Horní část odpovídá RTRBM. Tento model má 9 parametrů a jeho trénování vypadá následovně:

1. spočítej nový skrytý stav $\hat{\boldsymbol{h}}^{(t)}$ podle rovnice 3.8,



Obrázek 3.2: Schéma RNN-RBM modelu.

2. spočítej parametry RBM které závisí na $\hat{\boldsymbol{h}}^{(t-1)}$ a proveď CD-k pro nalezení modelem generovaného $\boldsymbol{v}^{(t)(i)}$,

3. vypočítej log-likelihood gradient cenové funkce C pro $\boldsymbol{W}$, $\boldsymbol{b}^{(t)}$ a $\boldsymbol{c}^{(t)}$,

4. propaguj gradienty zpětně v čase do RNN.

Získané gradienty lze použít k optimalizaci. Při té se snažíme minimalizovat log-věrohodnost ceny $C \equiv -\log P(\boldsymbol{v}^{(t)})$. Gradienty ve třetím kroku odpovídají rovnicím 4.3, 4.4 a 4.5.

Propagace gradientů v čase byla již popsána v 3.1.1, kde zásadním vztahem je rovnice 3.6. Máme-li spočítané gradienty RBM podle $\boldsymbol{W}$, $\boldsymbol{b}$ a $\boldsymbol{c}$, propagují se v čase přes spojovací váhy:

$$\frac{\partial C}{\partial \boldsymbol{W}'} = \sum_{t=1}^{T} \frac{\partial C}{\partial \boldsymbol{c}^{(t)}} \otimes \hat{\boldsymbol{h}}^{(t-1)}$$

$$\frac{\partial C}{\partial \boldsymbol{W}''} = \sum_{t=1}^{T} \frac{\partial C}{\partial \boldsymbol{b}^{(t)}} \otimes \hat{\boldsymbol{h}}^{(t-1)}$$

Pro změnu skrytého stavu obyčejného RNN pak platí:

$$\frac{\partial C}{\partial \hat{\boldsymbol{h}}^{(t)}} = \boldsymbol{W}_3 \frac{\partial C}{\partial \hat{\boldsymbol{h}}^{(t+1)}} \hat{\boldsymbol{h}}^{(t+1)}(1 - \hat{\boldsymbol{h}}^{(t+1)}) + \boldsymbol{W}' \frac{\partial C}{\partial \boldsymbol{c}^{(t+1)}} + \boldsymbol{W}'' \frac{\partial C}{\partial \boldsymbol{b}^{(t+1)}}$$

To platí pro $0 \leq t < T$. Počáteční vstup je nula a derivace $\partial C / \partial \hat{\boldsymbol{h}}^{(T)} = 0$. Pro úplnost ještě uvedu ještě derivace pro zbývající parametry:

$$\frac{\partial C}{\partial \boldsymbol{b}_{\hat{h}}} = \sum_{t=1}^{T} \frac{\partial C}{\partial \hat{\boldsymbol{h}}^{(t)}} \hat{\boldsymbol{h}}^{(t)}(1 - \hat{\boldsymbol{h}}^{(t)})$$

$$\frac{\partial C}{\partial \boldsymbol{W}_3} = \sum_{t=1}^{T} \frac{\partial C}{\partial \hat{\boldsymbol{h}}^{(t)}} \hat{\boldsymbol{h}}^{(t)}(1 - \hat{\boldsymbol{h}}^{(t)}) \otimes \hat{\boldsymbol{h}}^{(t-1)}$$

$$\frac{\partial C}{\partial \boldsymbol{W}_2} = \sum_{t=1}^{T} \frac{\partial C}{\partial \hat{\boldsymbol{h}}^{(t)}} \hat{\boldsymbol{h}}^{(t)}(1 - \hat{\boldsymbol{h}}^{(t)}) \otimes \boldsymbol{v}^{(t)}$$

Já jsem je ve své aplikaci nevyužil, protože jsou již implementované jako běžný modul.

Důležitou součástí trénování je také inicializace parametrů. To může vylepšit výslednou kvalitu modelu. Nejdříve jsem tedy natrénoval samostatné RBM. Tomu se věnuje celá kapitola 4. Další váhy se většinou nastaví na malé hodnoty, aby nevznikala ostrá symetrie. Sekvenční RNN-RBM se tak bude chovat ze začátku jako samostatná RBM, generující nějaké akordy a až později začne zachytávat temporální závislosti.

3.4 LSTM

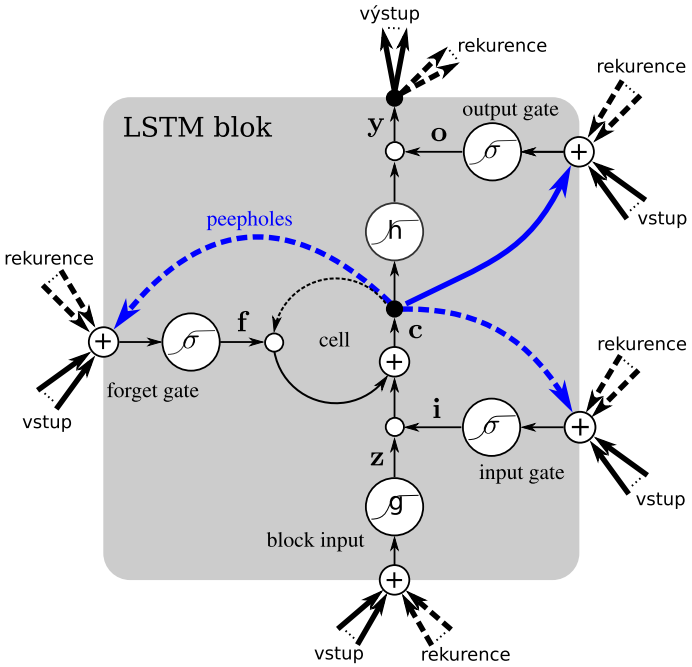
Tento model navrhli již Hochreiter – Schmidhuber (1997). Hlavní motivací bylo vyřešení nestabilních gradientů, které již jsem naznačil v kapitole 3.1.2. To je dosaženo pomocí efektivního návrhu architektury vynucující konstantní proud chyby přes její vnitřní stavy. Tato architektura se umí rozhodnout jak vstup ovlivní její *cell state*.

Bez ohledu na vnitřní strukturu, můžem brát tento modul jako RNN krabičku, která očekává na vstup $\boldsymbol{x}_t$ a $\boldsymbol{s}_{t-1}$ a na výstup dává $\boldsymbol{s}_t$. Díky tomu lze velmi snadno tyto krabičky zaměňovat. Můžeme také pohlížet na RNN jako speciální případ LSTM.

LSTM (obrázek 3.3) tedy obsahuje tzv. *input gate*, který rozhoduje, co si vezme ze vstupu, *output gate*, který zase snižuje pertubace chyby výstupu do dalších bloků a také *forget gate*, který rozhoduje o setrvání ve stavu. Říká se jim brány, protože sigmoida omezí vstup na $\langle 0, 1 \rangle$ a vynásobením jiným vektorem tak rozhoduje, co ze vstupu projde dál.

Základním kamenem je *cell state*. Ten se uchovává po celou dobu a je ovlivněn jen několika lineárními operacemi. Operace $*$ je násobení po složkách:

$$C^{(t)} = f^{(t)} * C^{(t-1)} + i^{(t)} * \tilde{C}^{(t)} \tag{3.9}$$



Obrázek 3.3: Architektura LSTM modulu.

Prvním krokem je *forget gate*. Pokud se objevý nějaký nový důležitý vstup, je třeba zapomenout ten starý:

$$f^{(t)} = \sigma(\boldsymbol{W}_f \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_f \boldsymbol{x}^{(t)}) \tag{3.10}$$

Dále potřebujeme rozhodnout jakou novou informaci přijmout. To má dvě fáze. První je *input gate* a pak tanh vrstva která spočítá konkrétní hodnotu $\tilde{C}^{(t)}$:

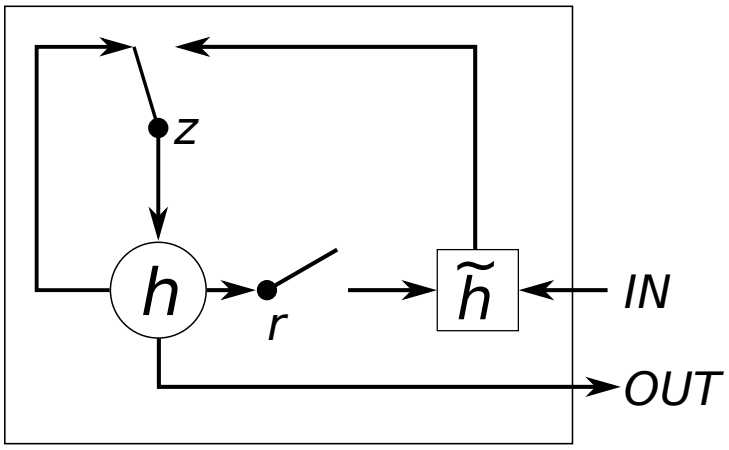
$$i^{(t)} = \sigma(\boldsymbol{W}_i \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_i \boldsymbol{x}^{(t)} + \boldsymbol{b}_i) \tag{3.11}$$

$$\tilde{C}^{(t)} = \tanh(\boldsymbol{W}_C \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_C \boldsymbol{x}^{(t)} + \boldsymbol{b}_C) \tag{3.12}$$

Z posledních tří rovnic zjistíme slíbený *cell state* v rovnici 3.9.

Nakonec spočítáme, co půjde na výstup. Ten bude založen na skrytém stavu, ale lehce filtrován. Výstupní vrstva opět bere vstup, skrytý stav a bias. Sigmoida určí jaké prvky půjdou na výstup. Nulová hodnota brány jednoduše změnu zamítne a jednička celou propustí. Cell state je transformován pomocí hyperbolického tangensu na $\langle -1, 1 \rangle$. Díky tomu může ovlivnit výstup na obě strany. Rovnice jsou následující:

$$o^{(t)} = \sigma(\boldsymbol{W}_o \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_o \boldsymbol{x}^{(t)} + \boldsymbol{b}_o) \tag{3.13}$$



Obrázek 3.4: Architektura GRU modulu.

$$\boldsymbol{h}^{(t)} = o_{(t)} * \tanh(C^{(t)}) \tag{3.14}$$

LSTM má také mnoho variant jako třeba *peephole connections*. Jejich podrobné porovnání vzniklo nedávno v článku od Greff et al. (2015).

3.5 GRU

Jeho idea je velmi podobná LSTM (Chung et al., 2014):

$$\boldsymbol{z}^{(t)} = \sigma(\boldsymbol{W}_z\boldsymbol{x}^{(t)} + \boldsymbol{U}_z\boldsymbol{h}^{(t-1)}) \tag{3.15}$$
$$\boldsymbol{r}^{(t)} = \sigma(\boldsymbol{W}_r\boldsymbol{x}^{(t)} + \boldsymbol{U}_r\boldsymbol{h}^{(t-1)}) \tag{3.16}$$
$$\tilde{\boldsymbol{h}}^{(t)} = \tanh(\boldsymbol{W}_h\boldsymbol{x}^{(t)} + \boldsymbol{U}_h(\boldsymbol{r}^{(t)} * \boldsymbol{h}^{(t-1)})) \tag{3.17}$$
$$\boldsymbol{h}^{(t)} = (1 - \boldsymbol{z}^{(t)}) * \boldsymbol{h}^{(t-1)} + \boldsymbol{z}^{(t)} * \tilde{\boldsymbol{h}}^{(t)} \tag{3.18}$$

Pro lepší pochopení příkládám známý obrázek 3.4. Liší se od LSTM tím, že nemá *cell state* a má pouze dvě brány a to *resetovací* $\boldsymbol{r}$ (angl. reset gate) a *aktualizační* $\boldsymbol{z}$ (angl. update gate). Resetovací brána určuje jak zkombinovat vstup s předchozím skrytým stavem a aktualizační brána co zachovat ze skrytého stavu a vstupu. Input a forget z LSTM jsou tedy spojeny do jedné aktualizační brány.

Tato architektura je poměrně nová a zatím není zcela prozkoumána. Nedá se říci, která je lepší. Každá úloha je trochu specifická a ukazuje se, že obě si vedou dobře. Mnohem důležitější než architektura je volba metaparametrů jako třeba počet skrytých uzlů. GRU slibuje větší rychlost, protože dělá méně počítání, ale na druhou stranu LSTM má větší možnosti, co se týká vyjadřovací schopnosti.

4 Optimalizační metody a učení

4.1 Gradientní metody

Jedná se o významnou kategorii optimalizačních technik. Jde o to najít parametry systému $\boldsymbol{\theta}$, aby vhodně modeloval vstupní vzory a minimalizoval chybu E. Změna parametrů se provádí podle následujícího vztahu:

$$\boldsymbol{\theta}(t) = \boldsymbol{\theta}(t-1) - \lambda \frac{\partial E}{\partial \boldsymbol{\theta}}. \tag{4.1}$$

Konstanta λ je parametr učení (angl. learning rate). Matematicky se jedná o délku kroku. Obvykle se v průběhu učení zmenšuje. Aktualizace parametrů by měly být menší o faktor 10^{-3} než rozsah hodnot parametrů.

Odečítání může být nahrazeno sčítáním. Je třeba si dobře zjistit, jakým směrem máme derivaci definovanou. Jestliže je derivace definována jako záporná, jako například v mém případě rovnice 4.3, pak je třeba otočit i znaménko sestupu a sčítat. Pokud je gradient obrácený, projeví se to většinou explozí chyby i parametrů.

Chyba nebo přesněji cenová funkce je velmi důležitou funkcí. Liší se podle toho, o jakou úlohu se jedná. V praxi nás ale nezajímá jak je model dobrý pro trénovací data, ale jak si poradí s příklady, které nezná. Tomu se říká testovací vzorek (angl. test set).

Důležitou gradientní metodou pro neuronové sítě je *Zpětná propagace* (angl. backpropagation). Tu jsem již zhruba popsal v kapitole 2.4. Výsledná chyba se řetězově aplikuje na všechny vrstvy v opačném pořadí než dopředný průchod. Bohužel není zaručeno, že taková metoda konverguje k optimu a může být velmi pomalá. Proto se používá mnoho triků, aby se učení zlepšilo, které popíši v následujících kapitolách.

Metoda může být buď stochastická nebo dávková. Obě mají své výhody a závisí opět na vstupních datech. Stochastické učení je výhodné v tom, že nepotřebuje k určení gradientu všechna data. Trénovací vzorek je náhodně zvolen a vyhodnocen. Z toho plyne, že je to rychlé a snadno se implementuje. Jeho nevýhodou je volba metaparametrů, a proto je třeba spustit trénování několikrát pro jejich nalezení. Tato metoda je z principu sekvenční a špatně se distribuje v Clusteru. Nejznámějším představitelem je *Stochastic Gradient Descent* – SGD. V poslední době se stochastickému učení věnují Bayer –

Osendorfer (2014).

Na druhou stranu dávkové učení má lepší podmínky ke konvergenci, některé metody urychlování jsou na něm založené. V dnešní době lze s výhodou využít grafické akcelerátory GPU k urychlení maticových operací. Nejznámějším představitelem je metoda sdružených gradientů (angl. Conjugate Gradients – CG).

Podrobné porovnání vytvořili v Ngiam et al. (2011), kde je podrobný rozbor trénování autoenkodérů pomocí SGD, CG a dalších. Ukazuje se, že SGD je neoptimální na mnoho úloh.

4.2 Kontrastivní divergence

V této kapitole představím zásadní metodu výpočtu parametrů RBM, která se jmenuje *Contrastive Divergence* – CD. Hinton (2002) ukázuje, že lze získat gradient ztrátové funkce l energetického modelu jako KL (Kullback–Leibler) divergence. Ta měří rozdíl mezi dvěma pravděpodobnostními rozděleními. Měříme podobnost očekávaných vstupních dat s modelem. Data modelu (zápornou divergenci) získáme pomocí Gibbsova vzorkování:

$$\nabla_{\boldsymbol{W}} l \approx \mathbb{E}[\boldsymbol{v}\boldsymbol{h}^T|\boldsymbol{v}_1] - \mathbb{E}^{(i)}[\boldsymbol{v}\boldsymbol{h}^T], \tag{4.2}$$

kde $\boldsymbol{v}_1$ odpovídá vstupnímu vektoru z trénovací množiny. Výpočet očekávané hodnoty pro RBM je v rovnicích 2.11 a 2.10. Očekávaná hodnota $\mathbb{E}^{(i)}$ z kroku (i) vznikne K dlouhým Gibbsovo řetězcem mezi viditelným a skrytým blokem. Jedná se o metodu *Markov Chain Monte Carlo* – MCMC. Délka takového řetězce plně dostačuje pro $K = 1$ (Bengio, 2009), ale vyšší k není na škodu. CD-k tak aproximuje Markovský řetězec kolem trénovacího bodu $\boldsymbol{v}_1$. Pro zvětšující i se tak více tomuto počátečnímu bodu vzdaluje. Změna parametrů podle této diference bude tak snižovat volnou energii trénovacích vzorků (to by mělo zvýšit věrohodnost) a zvýšit energii generovanému $\boldsymbol{v}^{(i)}$.

Při výpočtu očekávaných hodnot předpokládáme aproximaci:

$$\mathbb{E}^{(i)}[\boldsymbol{v}\boldsymbol{h}^T] \approx \boldsymbol{v}^{(i)}(\boldsymbol{h}^{(i)})^T.$$

Díky tomu pak můžeme zapsat konkrétní gradienty změny pravděpodobnosti konfigurace:

$$-\frac{\partial \log p(v)}{\partial \boldsymbol{W}} = \boldsymbol{v}_1(\boldsymbol{\mu})^T - \boldsymbol{v}^{(i)}(\boldsymbol{\mu}^{(i)})^T, \tag{4.3}$$

$$-\frac{\partial \log p(v)}{\partial \boldsymbol{c}} = \boldsymbol{\mu} - \boldsymbol{\mu}^{(i)}, \tag{4.4}$$

$$-\frac{\partial \log p(v)}{\partial \boldsymbol{b}} = \boldsymbol{v}_1 - \boldsymbol{v}^{(i)}. \tag{4.5}$$

V těchto rovnicích jsem použil vektor $\boldsymbol{\mu}$ pro označení očekávaných pravděpodobností aktivací ve skrytém bloku. To odpovídá rovnici 2.10. Podstatný rozdíl je v tom, že neprovádím vzorkování na binární hodnoty u posledního kroku CD-k. Binární stavy nám zaručují požadované úzké hrdlo, a proto se používají uvnitř CD-k, ale v posledním kroku (i) se musí použít pravděpodobnosti (Hinton, 2012). Gradient totiž nemůže záviset na šumu ze vzorkování.

Tyto rovnice je možné najít v různých obdobách v každé literatuře, ale málokdy jsou zapsány vektorově. Užitečný byl tak rozbor učení RBM od Schwehn (2010).

Důležité je také udělat vhodnou inicializaci. Obvykle se inicializují váhy $\boldsymbol{W}$ z normálního rozdělení se střední hodnotou 0 a malým rozptylem (např. $0,08$). Bias $\boldsymbol{b}$ a $\boldsymbol{c}$ je na začátku nulový. Zápornou inicializací $\boldsymbol{c}$ lze dosáhnout také řídkosti, ale na to jsem použil lepší nástroj v kapitole 4.5.

vstup : vstupní mini-batch $\boldsymbol{v}_1$ o velikost B a počet kroků k
výstup: výsledné gradienty $\nabla \boldsymbol{W}$, $\nabla \boldsymbol{b}$ a $\nabla \boldsymbol{c}$
spočítej $\boldsymbol{\mu}_1 = \mathbb{E}[\boldsymbol{h}|\boldsymbol{v}_1, \boldsymbol{\theta}]$;
$\boldsymbol{v}^{(0)} = \boldsymbol{v}_1$;
for $i \leftarrow 1$ **to** k **do**
  vzorkuj $\boldsymbol{h}^{(i-1)} \sim p(\boldsymbol{h}|\boldsymbol{v}^{(i-1)}, \boldsymbol{\theta})$;
  vzorkuj $\boldsymbol{v}^{(i)} \sim p(\boldsymbol{v}|\boldsymbol{h}^{(i-1)}, \boldsymbol{\theta})$;
  spočítej $\boldsymbol{\mu}^{(i)} = \mathbb{E}[\boldsymbol{h}|\boldsymbol{v}^{(i)}, \boldsymbol{\theta}]$;
end
$-\nabla W = (\boldsymbol{\mu}_1^T \boldsymbol{v}_1 - (\boldsymbol{\mu}^{(k)})^T \boldsymbol{v}^{(k)})/B$;
$-\nabla b = \texttt{mean}(\boldsymbol{v}_1 - \boldsymbol{v}^{(k)}, 1)$;
$-\nabla c = \texttt{mean}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}^{(k)}, 1)$;

Algoritmus 1: pseudokód funkce CD-k

Algoritmus 1 výpočtu CD-k vycházel z kódu od (Murphy, 2012), ale dodal jsem také výpočet derivace biasů a zobecnil na vektorové operace v *mini-batch*. Použil jsem tam funkci `mean`, která nedělá nic jiného než spočítá průměr všech sloupců. To odpovídá funkci `torch.mean`. Tato funkce bude volána pro každou *mini-batch* z trénovací množiny a epochu.

Parametrem takové funkce tedy bude vstupní *mini-batch* a počet kroků k Gibbs vzorkování. Výsledné gradienty jsou záporné tak, aby odpovídaly gradientnímu sestupu v rovnici 4.1.

Lze je samozřejmě aplikovat tak jak jsou, ale to v praxi vůbec nefunguje. Je třeba použít alespoň moment. Dále je možné ještě rozšířit o weight-decay nebo řídkost.

Další možností je *Persistentní CD*. Ten využívá vlastnosti, že se parametry moc nemění a omezuje vzorkování v každé iteraci. Algoritmus funguje většinou dobře, ale pokud se mění parametry hodně, konverguje CD-k rychleji.

4.3 Weight-decay

Weight-decay přičítá hodnoty k normálnímu gradientu. Jedná se o běžnou regularizaci, která penalizuje velké váhy. To snižuje přetrénování a vytváří lépe interpretovatelné váhy tím, že vynuluje nepotřebné koeficienty. Tlumí také hodnoty vzniklé na začátku trénování, kterých je třeba se zbavit:

$$\mathrm{L2} = \beta \frac{1}{2} \sum_i \theta_i^2 \tag{4.6}$$

$$\mathrm{L1} = \beta \sum_i |\theta_i| \tag{4.7}$$

L1 drží většinu koeficientů nulových a některé zas velké. Není to ale to samé co dělá řídkost v následující kapitole. Záleží na trénovaných datech, ale většinou se používá L1. Tato regularizace se aplikuje jen na váhy.

4.4 Moment

Rychlost změny parametrů je dána parametrem učení, ale ten nepomůže, pokud se gradient dostane do dlouhé ploché části, nebo do tzv. údolí optimalizované funkce. Moment totiž funguje jako těžká kulička, která nesleduje jen směr gradientu, jehož přímé sledování může způsobit cik-cak efekt. Taková optimalizace pak zamrzne na stejném místě.

Moment zavádí rychlost. Ta se upravuje pomocí gradientu a je třeba si ji pamatovat po celou dobu jedné epochy. Rychlost koule zpomaluje s časem. Tuto rychlost určuje parametr α. Její tvar je:

$$\nabla \boldsymbol{\theta}(t) = \boldsymbol{v}(t) = \alpha \boldsymbol{v}(t-1) - \lambda \frac{\partial E(t)}{\partial \boldsymbol{\theta}} \tag{4.8}$$

4.5 Řídkost

Motivací pro řídké parametry je lepší interpretovatelnost aktivací. Jestliže je aktivována většina uzlů pro všechny vstupy, budou takové reprezentace spíše šum a moc informace nepřidají. To můžeme sledovat pomocí histogramu středních aktivací skrytého uzlu jedné mini-batch.

Proto zavádíme tzv. řídkost (angl. sparsity target). Jejím parametrem je cílová pravděpodobnost t aktivace skrytých uzlů. Aktuální pravděpodobnosti získáme pomocí středních aktivací jedné mini-batch:

$$\boldsymbol{q}_{\text{nyní}}(\boldsymbol{v}) = \frac{1}{B}\sum_{b}\sigma(\boldsymbol{v}\boldsymbol{W}^T + \boldsymbol{c})_{bh}. \tag{4.9}$$

Potom definujeme penalizační funkci:

$$\boldsymbol{p}(\boldsymbol{q}) = -t\log(\boldsymbol{q}) - (1-t)\log(1-\boldsymbol{q}). \tag{4.10}$$

Jakmile jsou spočítané průměrné aktivace $\boldsymbol{q}_{\text{nyní}}$, potřebujeme spočítat novou hodnotu $\boldsymbol{q}_{\text{nové}}$. Pro zamezení velkých skoků se používá klouzavý průměr ve formě:

$$\boldsymbol{q}_{\text{nové}} = \lambda\boldsymbol{q}_{\text{staré}} + (1-\lambda)\boldsymbol{q}_{\text{nyní}}. \tag{4.11}$$

Derivace penalty vznikne z řetízkového pravidla:

$$\frac{\partial \boldsymbol{p}}{\partial \boldsymbol{W}} = \frac{\partial \boldsymbol{p}}{\partial \boldsymbol{q}}\frac{\partial \boldsymbol{q}}{\partial \boldsymbol{h}}\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}, \tag{4.12}$$

$$\boldsymbol{q} = \frac{1}{B}\sum_{b}\boldsymbol{h}_{bh}, \tag{4.13}$$

$$\boldsymbol{h} = \sigma(\boldsymbol{z}_{bh}), \tag{4.14}$$

$$\boldsymbol{z} = \boldsymbol{v}\boldsymbol{W}^T + \boldsymbol{c}. \tag{4.15}$$

Pak platí:

$$\frac{\partial \boldsymbol{p}}{\partial \boldsymbol{W}} = \frac{\boldsymbol{q}-t}{\boldsymbol{q}(1-\boldsymbol{q})}\boldsymbol{q}(1-\boldsymbol{q})\boldsymbol{v}^T = (\boldsymbol{q}-t)\boldsymbol{v}^T \tag{4.16}$$

Derivaci $\frac{\partial \boldsymbol{q}}{\partial \boldsymbol{h}} = 1/B$ jsem záměrně vynechal, protože střední hodnotu $\boldsymbol{q}$ normuji velikostí mini-batch jako u všech mini-batch výpočtů. Stejným způsobem pak získám derivaci pro bias $\boldsymbol{c}$:

$$\frac{\partial \boldsymbol{p}}{\partial \boldsymbol{c}} = \frac{\boldsymbol{q}-t}{\boldsymbol{q}(1-\boldsymbol{q})}\boldsymbol{q}(1-\boldsymbol{q}) = \boldsymbol{q}-t. \tag{4.17}$$

Je důležité aplikovat tuto penalizaci jak na váhy, tak na bias $\boldsymbol{c}$. Pokud bychom ji aplikovali jen na bias $\boldsymbol{c}$, mohlo by se stát, že bude bias klesat do záporných hodnot, aby tlumil skryté aktivace a váha naopak poroste do kladných hodnot, aby zvýšila jeho vliv.

4.6 Sledování průběhu učení RBM

Tato kapitola popisuje různé aspekty učení. Hodně věcí se může pokazit: od chyby v programu až po špatné metaparametry.

Základní charakteristikou je *rekonstrukční chyba*. Vzhledem k tomu, že používám CD-k, je výpočet jednoduchý. Odečtu v absolutní hodnotě vstupní hodnotu od rekonstruované. Průměr rozdílu je pak samotná chyba jako skalár.

Dále sleduji *ztrátovou funkci* l. Ta se odhaduje z cross-entropy (rovnice 2.15) [1] z trénovacího vstupu a rekonstrukce. Energetický model sice používá empirickou negativní logaritmickou věrohodnost, ale tu nelze při trénování stanovit kvůli vysoké výpočetní náročnosti:

$$l(\boldsymbol{\theta}, \mathcal{D}) = -\frac{1}{N} \sum_{\boldsymbol{v} \in \mathcal{D}} \log p(\boldsymbol{v}) \tag{4.18}$$

Pravděpodobnost vstupního vzorku vychází z rovnice 2.4:

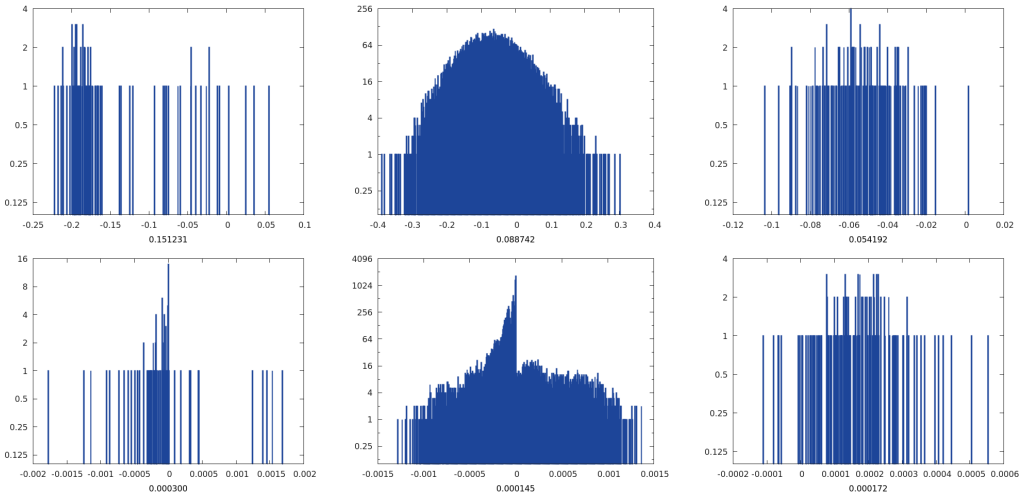
$$p(\boldsymbol{v}) = \sum_{\boldsymbol{h}} p(\boldsymbol{v}, \boldsymbol{h}|\boldsymbol{\theta}). \tag{4.19}$$

Vadí nám již známá dělící funkce 2.6. Specifickou vlastností RBM je *volná energie* v rovnici 2.12. V jazyce Lua/Torch vypadá její funkce:

```
function RBM:freeEnergy(visible)
    self.output:resize(self.n_hidden)
    self.output:copy(self.hbias)
    self.output:addmv(1, self.weight, visible)
    self.output:exp():add(1):log()
    local neg = self.output:sum()
    local pos = torch.dot(visible, self.vbias)

    return -neg-pos
end
```

Při trénování by obecně měla klesat, ale prakticky nevadí, když roste, protože to není optimalizovaná veličina. Optimalizuje se ztrátová funkce.

[1] Doporučen například ve známém tutorialu `http://deeplearning.net/tutorial/rbm.html`.



Obrázek 4.1: Histogramy hodnot a změn pro viditelný bias, váhy a skrytý bias v první epoše předtrénování RBM.

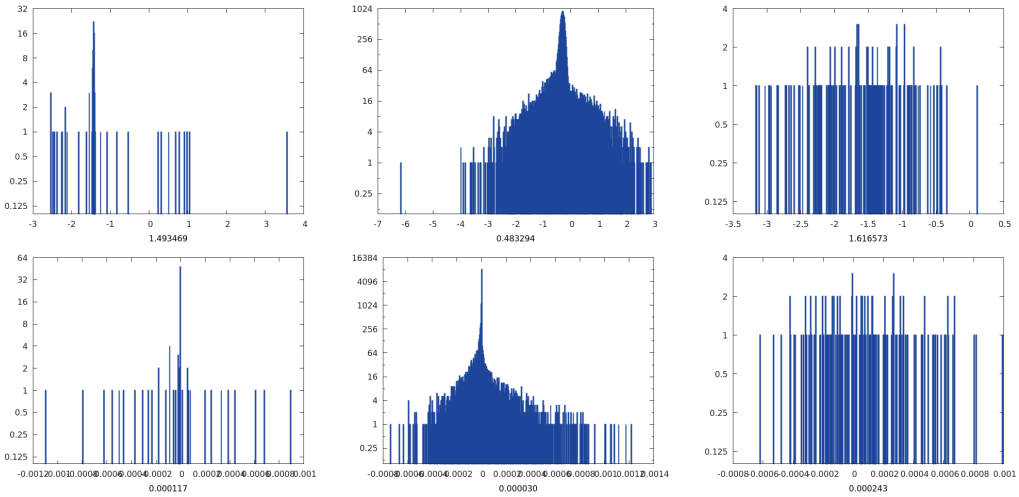
Volná energie se rovná ztrátové funkci děleno dělící funkce. Dělící funkce se v průběhu učení mění, a proto se mění i energie, a to může někdy velmi razantně. To se děje hlavně na začátku trénování. Hinton (2012) doporučuje používat volnou energii jako měřítko přeučení. Stejně tak jako rekonstrukce, tak i energie, by se neměla zásadně lišit pro trénovací data od testovacích.

V průběhu učení sleduji dále různé histogramy. V první řadě je to histogram biasů a vah. Histogramy by měly mít přibližně střední hodnotu v nule a nějaké kladné a záporné hodnoty. Na obrázku 4.1 je výsledek první iterace a na obrázku 4.2 z poslední. První řádka obsahuje hodnoty a druhá jsou jejich změny (rychlosti momentu). Pod každým histogramem hodnot je uvedena celková energie definovaná:

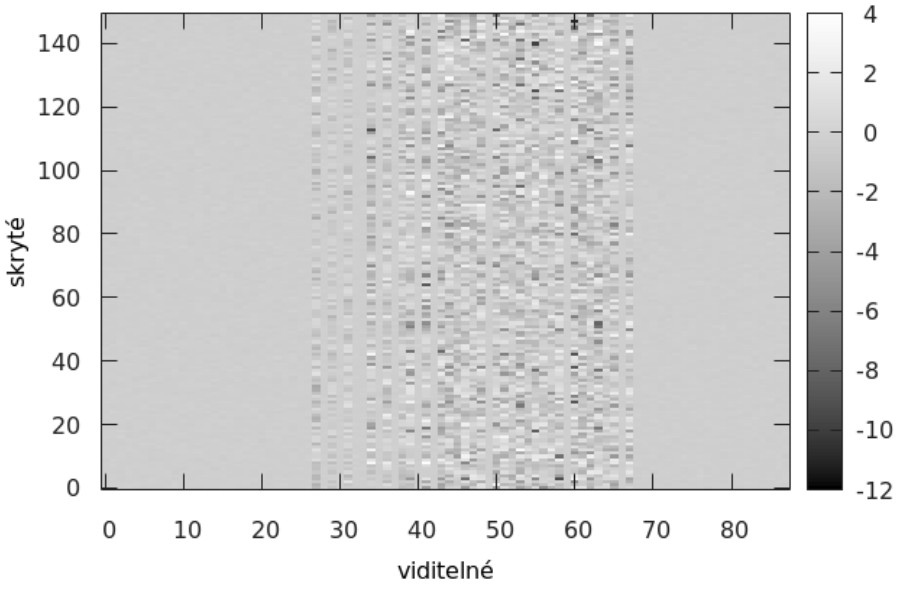
$$M = \sum_i |v_i|, \tag{4.20}$$

kde v_i jsou všechny sledované hodnoty v histogramu. Tato energie nesmí razantně růst ani klesat. Změny rychlostí by měla být menší o řád 10^{-3} než energie příslušných hodnot. Při použité metodě momentu popsaného v kapitole 4.4, je třeba sledovat tyto hodnoty a regulovat podle nich parametr učení.

V poslední řadě je také důležitý pohled na samotné váhy $\boldsymbol{W}$. Jejich řádky by měly odpovídat požadovaným hledaným vlastnostem. Při trénování na obrázcích jsou tyto vlastnosti zřetelně vidět jako hranové operátory. V našem případě jsou to různé kombinace intervalů tónů. Na obrázku 4.3 je výsledek 50. epochy předtrénování.



Obrázek 4.2: Histogramy hodnot a změn pro viditelný bias, váhy a skrytý bias v poslední epoše předtrénování RBM.



Obrázek 4.3: Předtrénovaná matice vah RBM po 50 epochách.

4.7 RMSProp

Oblíbenou optimalizační metodou pro učení rekurentní neuronové sítě je *RMSProp*. Jedná se také o gradientní metodu. Problémem gradientního sestupu je volba parametru učení. Některé gradienty mohou být velké a některé malé. To se může snadno změnit v průběhu trénování, a proto se parametr učení špatně volí.

Metoda *Rprop* využívá znaménka gradientu pro adaptivní změnu kroku pro každý gradient. Pokud se znaménko nezměnilo od posledního gradientu, zvětšuje se krok a naopak. Tento krok je omezen zhora a zdola. Problémem je, že tato metoda nefunguje pro mini-batch, protože výsledné váhy by byly příliš velké. Řekněme, že získáme gradient $+0.1$ na devíti po sobě jdoucích mini-batch. Tato hodnota se kumuluje víc a víc s velikostí mini-batch jako změna přírůstku k vahám. Následující gradient o velikosti -0.9 nevrátí hodnotu vah o 10 kroků zpátky.

Popsané problémy s mini-batch řeší RMSProp se stejnou robustností jako Rprop. Výsledný gradient se dělí jeho velikostí. Zavádí se klouzavý průměr kvadrátu gradientu:

$$\boldsymbol{g}^{(t)} = \lambda \boldsymbol{g}^{(t-1)} + (1-\lambda)\left(\frac{\partial E^{(t)}}{\partial \boldsymbol{W}}\right)^2 \tag{4.21}$$

Gradient $\frac{\partial E^{(t)}}{\partial \boldsymbol{W}}$ pak dělíme druhou odmocninou $\sqrt{\boldsymbol{g}^{(t)}}$. Na získaný gradient je třeba aplikovat parametr učení. Dále je možné aplikovat různé momenty, ale to je zatím předmětem výzkumu.

Hinton doporučuje používat tuto metodu na velké redundantní vstupy a rekurentní sítě. To je přesně můj případ.

5 Podrobný popis řešení

5.1 Možnosti knihovny Torch

V praktické části jsem používal vědecký výpočetní framework *Torch7*[1]. Podporuje celou řadu algoritmů strojového učení, kód lze snadno přenést na GPU, vyniká vysokou rychlostí[2] a je založen na neobvyklém jazyce *Lua*. Má velmi úzké propojení s C a matematickou knihovnou *BLAS*. K dispozici je mnoho rozšiřujících modulů. Nejdůležitější modul byl `rnn` od autorů Léonard et al. (2015). Není tak rozšířen jako *Python/Theano*, ale stojí za ním hlavně *Facebook* nebo *Google DeepMind*[3].

Jazyk *Lua* má některá specifika. Je založena na *metamechanismu*. Jedná se o obdobu přetěžování z jiných jazyků. Každý objekt má svojí *metatabulku*, do které lze přidat libovolnou *metametodu*. Každá metoda je tak přiřazena k určité události jako součet nebo porovnání. K tomu slouží funkce `setmetatable()`:

```
1  local x = {value = 5}
2
3  local mt = {
4      __add = function (lhs, rhs)
5          return { value = lhs.value + rhs.value }
6      end
7  }
8
9  setmetatable(x, mt)
10
11 local y = x + x
12 print(y.value) -- prints 10
```

V základu například neposkytuje objekty, ale není problém je vytvořit. Příjemnou věcí na Torch je, že se programátor nemusí starat o paralelismus. Kód běží v případě potřeby paralelně díky *OpenMP*. Paralelismus se dá nastavit pomocí proměnné prostředí `OMP_NUM_THREADS`.

Pro vývoj jsem používal hodně *Jupyter* notebook[4]. Ten je známý především ze světa Pythonu, ale existují kernely pro spoustu dalších jazyků včetně *Lua*.

[1] `http://torch.ch`
[2] `https://attractivechaos.github.io/plb`
[3] Na konci dubna 2016 přešel Google na svůj framework *TensorFlow*.
[4] `http://jupyter.org`

5.2 Práce s MIDI

Rozhraní MIDI má dlouhou historii. Jeho původním využitím je přenos a interpretace událostí mezi různými zařízeními. Není tedy nijak omezené pouze na hudbu.

Formát MIDI může být typu 0, a tedy všechny události jsou v jedné stopě. Typ 1 přinesl oddělené stopy (track). Rozhraní *General MIDI* popisuje události, parametry a také stanovuje seznam nástrojů. Každá stopa je rozdělena do 16 kanálů, přičemž 10. kanál je vyhrazen bicím nástrojům. Existují určitá rozšíření, ale já jsem počítal se základními 128 nástroji.

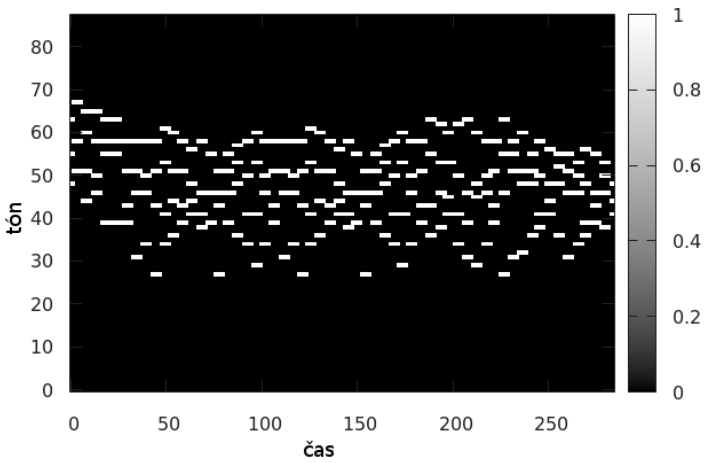
Každá událost má svojí časovou značku. Ta se měří v tikách od poslední události. Tempo se měří jinak, než jsou muzikanti zvyklí, a to v počtu mikrosekund na jednu čtvrťovou notu. Délka čtvrťové noty v tikách je uvedena na začátku souboru a událost „změna tempa“ se může odehrát kdykoli.

Pro načítání MIDI jsem použil dostupnou knihovnu MIDI.lua. Ta poskytuje dva různé režimy načtení souboru. Opus zachovává originální strukturu souboru. Každá událost se měří v časech od předchozí události. Tón je tedy složen z události začátku a konce. Také je možné použít režim score, kde jsou události tónů sjednoceny do jedné události. Tento režim byl plně vyhovující. Jeho formát je následující:

```
local my_score = {
    96,
    {
        {'patch_change', 0, 1, 8},
        {'set_tempo', 0, 1200000}
        {'note', 5, 96, 1, 25, 98},
        {'note', 101, 96, 1, 29, 98},
    }
 }
```

První položka tabulky znamená počet tiků na jednu čtvrťovou notu. Další položky jsou jednotlivé stopy. Každá stopa obsahuje seznam událostí. Ty mohou být v libovolném pořadí. Co znamenají konkrétní čísla je v dokumentaci `http://www.pjb.com.au/comp/lua/MIDI.html`. První číslo je ale typicky čas. V této ukázce jsem například vytvořil událost v čase 0, pro změnu nástroje 8 na kanálu 1 a změnil tempo na 1200000 μs za čtvrťovou notu a dále jsem vytvořil dva tóny.

Partituru jsem vytvořil tak, že jsem nejdříve normalizoval délky not a časy událostí podle nejkratší požadované noty (dvaatřicetinové) a pak jsem vkládal jedničky na pozice v tenzoru tak, že začínaly na pozici dané začátkem události a jejich délce. Výsledek je například na obrázku 5.1. Taková



Obrázek 5.1: Trénovací data ve formě partitury.

reprezentace je vhodná jako vstup neuronové sítě. Dále jsem omezil rozsah dostupných výšek not. MIDI předepisuje až 128 tónů, ale to není využito. Rozsah piana je od MIDI čísla 21 do 108. Celkově to je 88 tónů. Dále jsem se zbavoval rytmického doprovodu. Ten se jednoznačně odlišuje tím, že je v kanálu 10, a to v kterékoli stopě.

Pro potřeby rekurentní sítě jsem musel načítat do tabulky o počtu časových kroků pro BPTT tensory o velikosti $B \times V$, kde B je velikost mini-batch a V je velikost vstupu. To je blíže popsáno na webu knihovny `rnn` na stránce `https://github.com/Element-Research/rnn`.

5.3 RBM modul pro `torch.nn`

Abych mohl využit možností rekurentního modulu `rnn`, implementoval jsem chybějící implementaci RBM tak, aby splňovala rozhraní `nn.Module`. Vytvořil jsem následující mapování:

- `updateOutput` provede Gibbs vzorkování a vrací hodnotu `self.vt`,
- `updateGradInput` vypočítá gradient cenové funkce vzhledem k $\boldsymbol{W}$, $\boldsymbol{b}$ a $\boldsymbol{c}$,
- `reset` gradient vah $\boldsymbol{W}$ inicializuje z normálního rozdělení s parametry $\mu = 0$ a $\sigma = 0,08$. A také $\boldsymbol{b} = 0$, $\boldsymbol{c} = 0$.

Uvnitř kódu je rozdělená rozpoznávací a generující fáze pro vnitřní potřeby, ale funkce jsou případně k dispozici.

Kód je napsán tak, aby fungoval jak v mini-batch módu, tak samostatně. Dále je možné dát na vstup tabulku pro rekurentní zapojení, které je ve formátu `{input(t), vbiasadd(t), hbiasadd(t)}`. Vstup je jeden časový moment z partitury a dodatečné biasy pochází například z předchozích `nn.LineasNoBias`.

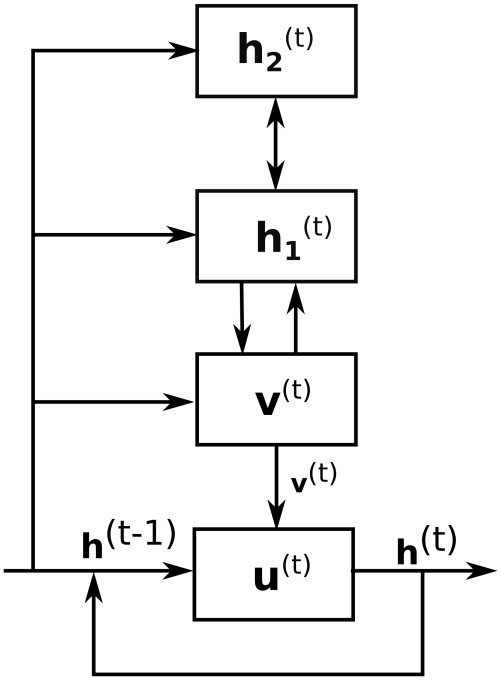
5.4 Implementace sítě

Teoreticky jsem popsal architekturu RNN-RBM v kapitole 3.3. Prakticky můžu libovolně nahradit RNN za kterýkoli rekurentní modul. Celá síť bohužel nejde uspořádat jako jedna sekvence, protože by to porušilo základní pravidlo rekurentního vztahu. Ten má očekávat vstupní tabulku `{input(t), output(t-1)}` a na výstup dát `{output(t)}`. Tyto hodnoty je třeba nejdříve spočítat a pak je dosadit do MLP obsahující RBM. Chyba RNN závisí na `input(t)` a na chybě biasů z času `t+1`. Kód je následující:

```
rnn_outputs, mlp_outputs, mlp_grads  = {}
-- 1)
for step=1,rho-1 do
    rnn_outputs[step] = rnn:forward(inputs[step])
end

-- 2)
for step=2,rho do
    mlp_outputs[step] = mlp:forward
          {inputs[step], rnn_outputs[step-1]}
end

-- 3)
for step=rho,2,-1 do
    mlp_grads[step] = mlp:backward
          ({inputs[step], rnn_outputs[step-1]}, nil)
end

-- 4)
for step=rho-1,1,-1 do
    rnn:backward(inputs[step], mlp_grads[step+1])
end
```

1. Udělám dopřednou propagaci přes skryté stavy rekurentní části. Tyto stavy nijak nezávisí na výstupu, ale jen vstupu a interních přechodech.

2. Na vstup `mlp` sekvence pošlu vstup a předchozí skrytý stav. To obsahuje vazby $\boldsymbol{W}'$ a $\boldsymbol{W}''$.



Obrázek 5.2: Architektura RNN-DBN.

3. Poté počítám zpětnou propagaci. Jejím výsledkem je gradient vzhledem k vstupním vazbám `mlp` modulu.

4. Získané gradienty dále předám do rekurentního modulu `rnn`.

Tento postup jsem odvodil z řešení od Boulanger-Lewandowski et al. (2012). V době psaní této práce je k dispozici také implementace pomocí knihovny *Theano*. Mé řešení je více modulární. Další řešení na obrázku 5.2 navrhuje Goel et al. (2014). Sám autor přiznává, že přidávání dalších vrstev RBM zásadně nezlepší výsledky. Zásadnější problém je základní RNN.

5.5 Předtrénování

Nejprve jsem rozdělil data na trénovací a testovací. Pro sledování průběhu trénování, jsem vytvořil funkce pro výpočet rekonstrukční chyby a volné energie. Rekonstrukční chyba se intuitivně počítá jako střední hodnota rozdílu predikované a vstupní hodnoty. Výpočet volné energie poskytuje rovnou třída RBM. Volná energie je skalár, takže je stačí sečíst pro všechna data.

Více o trénování RBM poskytuje Hinton (2012) nebo Yosinski – Lipson (2012). Některé techniky jsem popsal v kapitolách 4.3, 4.4 a 4.5.

	JSB Chorales	Piano-MIDI.de
parametr učení	0.018	**0,003**
moment	0,5	0,5
L1	0.0008	0.0008
zpomalení (decay-rate)	0,9	0,9
cílová pravděpodobnost (t)	0,08	0,08
cena řídkosti	0.0006	**0.000002**
SGD parametr učení	**0.002**	0,004
SGD zpomalení	0,97	0,97

Tabulka 5.1: Zvolené metaparametry pro testované kolekce. Tučné hodnoty jsou odlišné od přednastavených.

Trénování probíhá v následujících krocích:

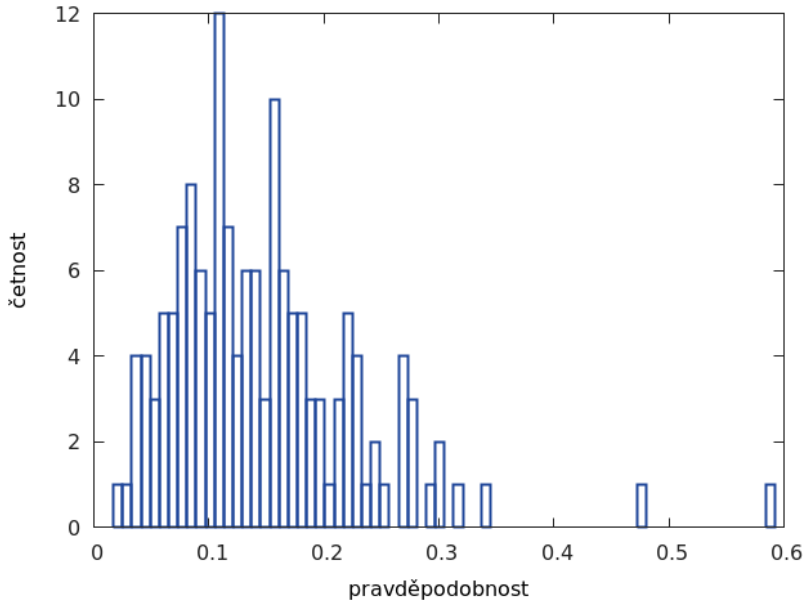
- načíst data,
- spočítat derivaci metodou CD-k (viz kapitola 4.2),
- aktualizace parametrů pomocí momentu,
- přičtení funkce řídkosti.

Při ladění parametrů jsem se držel zásad, popsaných v kapitole 4.6. Výsledné použité hodnoty jsou v tabulce 5.1.

Nejdříve jsem zrušil penalizaci řídkosti a ladil jen parametr učení. Začal jsem na nízké hodnotě v řádu 10^{-4} a postupně parametr zvyšoval. Při tom jsem sledoval, jak se mění matice vah. S vysokými hodnotami parametru učení klesala rekonstrukční chyba velmi rychle, ale po několika epochách se učení zastavilo.

Moment jsem na začátku nastavil na $0,5$, jak doporučuje Hinton (2012). Tato hodnota dostatečně zpomalovala změny gradientu v počátku trénování. V půlce učení zvyšuji moment na $0,7$ a pak ještě jednou až na hodnotu $0,9$. Po provedené změně se může rekonstrukční chyba mírně zhoršit, ale to je v pořádku.

Nastavení řídkosti bylo složitější. Když pominu zmatek v publikacích, ze kterých není jasné, jak se má vlastně aplikovat, tak má i více parametrů. Praktičtější náhled je součástí článku od Nair – Hinton (2009). Jiné odvození napsal *Nghia Ho* na svém blogu `http://nghiaho.com/?p=1817`. Ten ale použil jinou penalizační funkci než Hinton a nefungovala dobře v krajních hodnotách.



Obrázek 5.3: Histogram pravděpodobností aktivací skrytých jednotek.

V kapitole 4.5 je podrobné odvození vztahů. Nejdříve spočítám průměrné aktivace z $\boldsymbol{\mu}_1$ získané v algoritmu 1, aplikuji na ně klouzavý průměr, spočtu derivace a odečtu je od příslušného o cíle.

Poté je třeba ladit $\boldsymbol{q}$ tak, aby bylo blízko t. To jsem dosáhl pomocí parametru ceny řídkosti. Dále je třeba sledovat samotné rozdělení průměrných aktivací jako na obrázku 5.3. Nesmí se stát, aby se rozdělení histogramu pohybovalo kolem cílové hodnoty. To by narušilo průběh učení. Pohyboval jsem tedy opět s cenou řídkosti. Tady musí nastoupit jistý kompromis mezi prvním a druhým požadavkem.

5.6 Generování

Natrénovaný model je sám od sebe generativní. Stačí stanovit počáteční stav stavu `rnn`. Výstup RBM v čase t závisí jen na předchozím skrytém stavu. Z toho se spočítá uvnitř `mlp` bias. Kód pro generování je následující:

```
1  piano_roll = torch.Tensor(length, n_visible)
2  zeros = torch.zeros(n_visible)
3
4  rnn_outputs={}
5  rnn_outputs[0] = rnn:forward(input_pool[torch.random(1,#input_pool)])
6
7  for t=1, length do
8    local sampled_v = mlp:forward{zeros, rnn_outputs[t-1]}
```

```
9     rnn_outputs[t] = rnn:forward(sampled_v)
10
11    piano_roll[t]:copy(sampled_v)
12  end
```

5.7 Obsluha aplikace

Pro spuštění knihovny *Troch7* je potřeba operační systém *Linux*, nebo *Mac OS X*. Distribuci *Torch7* lze získat podle návodu na stránce `http://torch.ch/docs/getting-started`. Tato instalace již nejde přemístit, ale může být kdekoli v uživatelské složce. Po kompilaci a instalaci je třeba aktivovat proměnné prostředí podle oficiálního návodu.

Já využívám vedle těch standardně nainstalovaných balíčků ještě tyto balíčky:

```
$ luarocks install rnn
$ luarocks install midi
$ luarocks install gnuplot
$ luarocks install optim
```

Následně je třeba stáhnout příslušnou kolekci MIDI. Já jsem používal kolekce `http://www-etud.iro.umontreal.ca/~boulanni/icml2012`.

Trénovací skript se jmenuje `train.lua`. Jeho seznam parametrů je následující:

```
Nastavení
  -data_dir
      adresář s trénovacími daty ve formátu MIDI. [train]
  -prefix    prefix všech výsledných souborů []
  -v         podrobný výpis [false]
Parametry modelu
  -n_hidden        Počet uzlů ve skryté vrstvě RBM. [150]
  -n_recurrent     Počet uzlů v rekurentním modulu. [100]
  -model           'lstm' nebo 'gru' [lstm]
  -init_rbm_from   načíst RBM ze souboru []
Optimalizační parametry
  -learning_rate   parametr učení [0.018]
  -momentum        moment [0.5]
  -L1              L1 regularizace [0.0008]
  -max_pretrain_epochs
      počet plných epoch při RBM přetrénování  [50]
```

```
  -sparsity_decay_rate  zpomalení řídkosti [0.9]
  -sparsity_target      cíl řídkosti [0.08]
  -sparsity_cost        cena řídkosti [0.0006]
  -sgd_learning_rate
      parametr učení pro SGD (RMSProp) [0.004]
  -sgd_learning_rate_decay
      zpomalení parametru učení [0.97]
  -sgd_learning_rate_decay_after
      v~jaké epoše začít snižovat parametr učení [40]
  -max_epochs           počet epoch trénování  [80]
  -rho                  počet časových kroků BPTT [32]
  -batch_size           velikost mini-batch [100]
  -stat_interval        interval statistik [256]
  -opencl               použít OpenCL [false]
  -cuda                 použít CUDA [false]
```

Trénováním vznikne několik souborů ve složce models. První je přetrénované RBM. To ukládám v polovině trénování a na konci. Je možné přeskočit předtrénování a použít jej pro trénování rekurentního modelu a to pomocí argumentu `-init_rbm_from`. Při tom ukládám každých deset epoch natrénované moduly.

Z natrénovaného modelu se vzorkuje skriptem `sample.lua` a má několik parametrů:

```
Nastavení
  -data_dir
      adresář s testovacími daty ve formátu MIDI. [test]
  -rnn_model   rekurentní modul [models/recurrence-rnn_10.dat]
  -mlp_model   MLP část modulu [models/recurrence-mlp_10.dat]
  -n_hidden    Počet uzlů ve skryté vrstvě RBM. [150]
  -n_recurrent Počet uzlů v rekurentním modulu. [100]
  -length      vzorkovaná délka [300]
  -o           výsledný MIDI soubor [sampled.mid]
```

6 Zhodnocení výsledků

6.1 Výpočetní složitost a měření

V této kapitole se budu věnovat konkrétním dopadům na výpočetní složitost a to nejdříve pro předtrénování, a pak doladění.

Všechna čísla byla uložena ve formátu `double`. Ukázalo se, že typování viditelného binárního vstupu jako `byte` přinášelo jen problémy. Binární stavy jsou jen speciální případ. Všechny ostatní tensory jsou reálná čísla z podstaty. Některé operace jako `log` jsou definovány jen pro `double`, a proto by bylo třeba mnoho přetypování se zanedbatelným výkonnostním nárůstem.

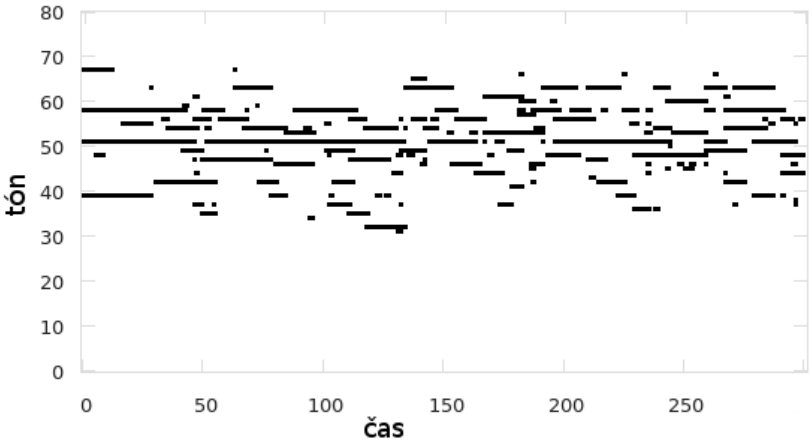
Předtrénování jsem prováděl nejvíce ve 100 epochách. V praxi bylo většinou kratší. V každé epoše jsem využil celou trénovací množinu. Ta obsahuje již načtené *mini-batch* vstupních vektorů. V kapitole 4.2 jsem popsal oblíbenou aproximaci ztrátové funkce RBM kontrastivní divergenci. Její délku jsem volil $k = 15$. To znamená zásadně méně počítání, než kdybych počítal empirický vztah 2.4. Gradientní sestup probíhal podle kapitoly 5.5.

Empiricky jsem zjistil, že je důležité vytvořit více skrytých uzlů než viditelných. Tím vznikne asociativní paměť s velkou kapacitou. Kvůli tomu je nutné řídit tyto aktivace pomocí cíle řídkosti v kapitole 4.5, aby nedocházelo k přeučení. Pro 88 vstupních uzlů jsem zvolil 150 skrytých uzlů. Na vytvoření úzkého hrdla bych potřeboval hlubokou síť, která by měla na vrcholu asociativní paměť. Vytvořit velké vrstvy a také velké *mini-batch* přináší znatelně větší nároky na procesor, ale *Torch* se přizpůsobí tím, že využije všechny procesory naplno. Dokáže tedy vyhodnotit, kdy se mu paralelismus vyplatí. Implementoval jsem také podporu pro *OpenCL*. Jedná se o otevřený standard pro paralelní programování. Díky tomu je možné provádět výpočet také na GPU.

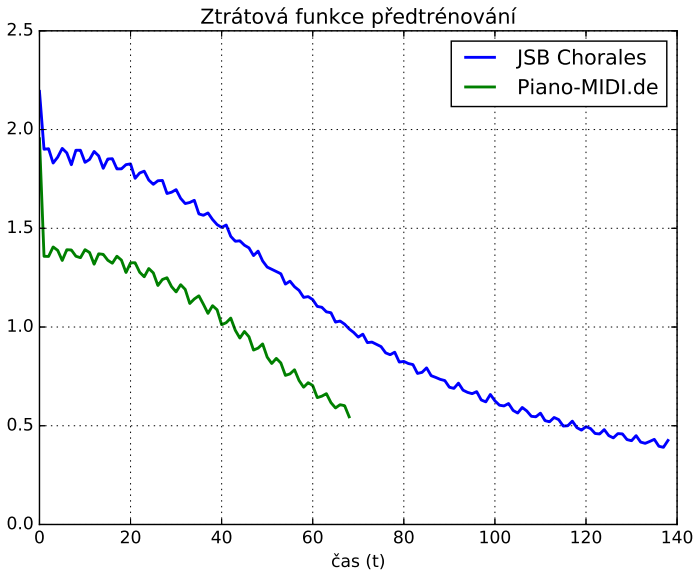
6.2 Předtrénování

Při předtrénování RBM jsem sledoval její ztrátovou funkci, jak byla popsána v kapitole 4.6, rekonstrukční chybu a volnou energii. Průběh ztrátové funkce pro použité kolekce je na obrázku 6.2.

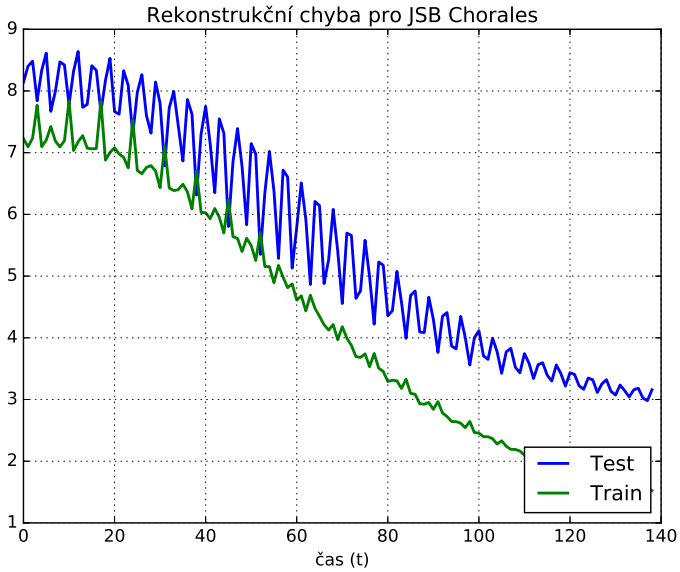
Rekonstrukční chyba by měla stejně jako ztrátová funkce klesat. Při změně momentu je možné, že se rekonstrukční chyba trochu zvedne. To jsem dělal přesně v polovině trénování a ve druhé třetině. Výsledek prů-



Obrázek 6.1: Ukázka generované partitury.



Obrázek 6.2: Průběh ztrátové funkce při předtrénování RBM.



Obrázek 6.3: Průběh rekonstrukční chyby při předtrénování RBM.

běhu rekonstrukční chyby je na obrázku 6.3, a to pro trénovací i testovací množinu.

Poslední statistikou je volná energie na obrázku 6.4. Uvádím zde jen grafy pro kolekci JSB Chorales.

6.3 Rekurentní model

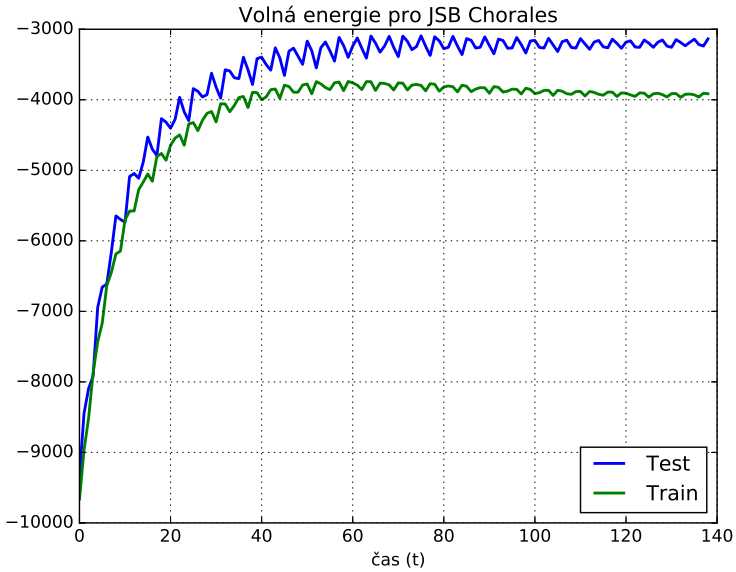
Následně jsem sestavil celou síť založenou na modelu LSTM-RBM a provedl její učení. Z implementačních důvodů jsem rozdělil LSTM na část, která řeší časové závislosti a RBM. Náročnost zpětné propagace v čase je úměrná zvolené délce paměti ρ. Ta může růst teoreticky do nekonečna, ale já jsem jí omezil na $\rho = 32$. V kapitole 5.4 jsem popsal podrobně implementaci trénování.

Pro trénování jsem použil následující známé kolekce:

JSB Chorales Barokní chorály od Johanna Sebestiana Bacha, vyznačující se dlouhými tóny s plnou harmonií a většinou v mollové tónině.

Piano-MIDI.de Nejznámější kolekce klasické hudby, která se stále rozšiřuje. `http://www.piano-midi.de`

Na těchto kolekcích se běžně testují různé architektury sítí. Jejich výkon se porovnává pomocí negativní logaritmické věrohodnosti $-\log l$ (angl.



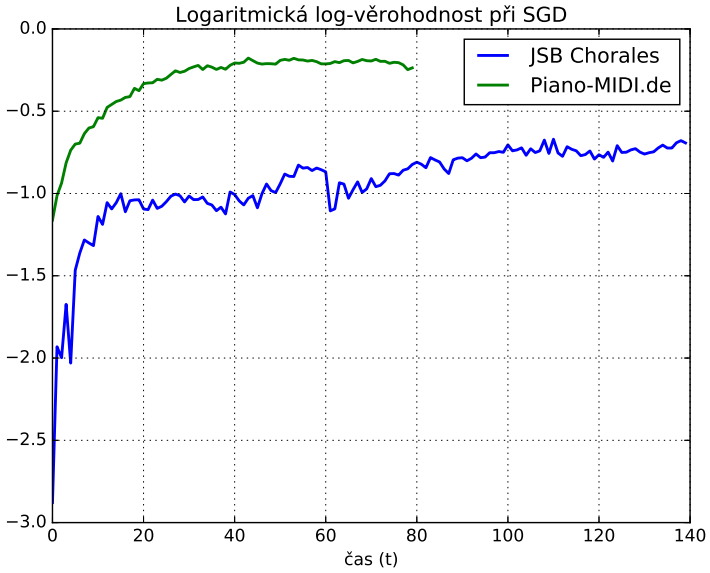
Obrázek 6.4: Průběh volné energie při předtrénování RBM.

	JSB Chorales	Piano-MIDI.de
$-\log l$	$-0,6883$	$-0,2366$
Precision [%]	$79,6745$	$97,3133$
Recall [%]	$71,1453$	$76,9727$
Accuracy [%]	$60,631$	$75,6698$
F-measure	$75,1687$	$85,9561$

Tabulka 6.1: Statistiky mého modelu z poslední epochy.

negative log-likelihood), která odpovídá cross-entropy (viz rovnice 2.15) predikované a přesné hodnoty. Výsledky pro poslední epochu jsou v tabulce 6.1. Celý průběh je na obrázku 6.5. Pro hodnocení kvality jsem použil metriky používané pro různé generativní modely, které navrhli například Sigtia et al. (2015). Pro porovnání s dalšími metodami v tabulce 6.2 jsem použil hodnoty od Boulanger-Lewandowski et al. (2012). U některých modelů jsou hodnoty uvedené jako N/A, protože je autor neuvádí.

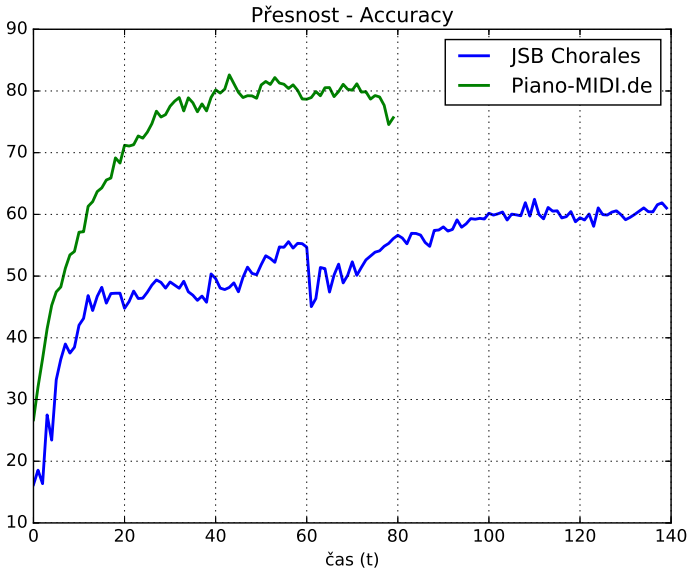
Porovnával jsem predikovaný vektor a trénovací. Vektor obsahuje jedničku, když daný tón má hrát. Z toho jsem zjistil počet platných výsledků (angl. true positive – TP) a chyby prvního a druhého typu (angl. false-positive – FP a false-negative – FN). Z toho lze spočítat následující ohodnocení:



Obrázek 6.5: Průběh ztrátové funkce při trénování rekurentní sítě.

Model	JSB Chorales		Piano-MIDI.de	
	$-\log l$	ACC [%]	$-\log l$	ACC [%]
GMM+HMM	$-11,89$	$19,24$	$-15,3$	$7,91$
MLP	$-8,70$	$30,41$	$-8,13$	$20,29$
RNN (HF)	$-8,58$	$29,41$	$-7,66$	$23,34$
RNN-RBM (HF)	$-6,27$	$33,12$	$-7,09$	$28,92$
RNN-DBN	$-5,68$	N/A	$-7,15$	N/A
RNN-NADE (HF)	$-5,56$	$32,50$	$-7,05$	$23,42$
LSTM-RBM	$\mathbf{-0,69}$	$\mathbf{60,631}$	$\mathbf{-0,24}$	$\mathbf{75,67}$
DBN-BLSTM	$-3,47$	N/A	-4,63	N/A

Tabulka 6.2: Porovnání s jinými modely. První sloupec je negativní logaritmická věrohodnost a druhý je přesnost. Má implementace LSTM-RBM je tučně.



Obrázek 6.6: Průběh přesnosti rekonstrukce při trénování rekurentní sítě.

$$\text{Accuracy} = \sum_{t=1}^{T} \frac{TP[t]}{TP[t] + FP[t] + FN[t]}, \tag{6.1}$$

$$\text{Precision} = \sum_{t=1}^{T} \frac{TP[t]}{TP[t] + FP[t]}, \tag{6.2}$$

$$\text{Recall} = \sum_{t=1}^{T} \frac{TP[t]}{TP[t] + FN[t]}, \tag{6.3}$$

$$\text{F-measure} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}, \tag{6.4}$$

kde T odpovídá počtu vzorků v trénovací množině. $TP[t]$ je počet správných predikcí při vstupu t. Tato čísla lze pak vydělit počtem T a vynásobit 100 pro získání hodnoty v procentech. Tyto metriky jsem měřil po každé epoše. Z těchto statistik jsem vybral graf přesnosti (angl. accuracy) na obrázku 6.6. Jakmile není ustálená optimalizovaná funkce, může dojít buď ke kmitání, nebo zastavení učení. To závisí také na charakteru vstupních dat. U kolekce JSB Chorale je vidět, co se stalo s rekonstrukční chybou při zvýšení momenta při 60. epoše.

7 Závěr

Tato práce ukázala možné řešení generování podkresové hudby pomocí rekurentních neuronových sítí za pomoci „učení bez učitele“. Vstupní a výstupní data byla ve formátu MIDI a vstupem neuronové sítě se stal vektor aktuálně znějících tónů v čase t. Neuronová síť měla „paměť“ ve formě LSTM modulu a výstupní RBM.

Představil jsem oblíbené temporální varianty jak grafických modelů, tak obecně neuronových sítí. Popsal jsem zpětnou propagaci v čase a problémy s ní spojené.

V této práci jsem dále ukázal některé aktuální metody modelování vícedimensionálních procesů. Ověřil jsem jejich značný potenciál v oblasti generování podkresové hudby. Neomezoval jsem se jen na jednohlasou hudbu, která by pro zadání jednoduché podkresové hudby stačila, ale zobecnil jsem řešení na polyfonní hudbu libovolné složitosti. Ukázalo se, že variabilita hudby komplikuje volbu metaparametrů a bylo by potřeba použít dalších metod automatického strojového učení.

Vytvořil jsem trénovací a vzorkovací skript v jazyce *Lua* s využitím vědeckého výpočetního frameworku *Torch7*. Tyto skripty je možné spouštět s různými parametry. Dále je možno použít k trénování jak rozhraní CUDA, tak OpenCL. Velikost vstupních dat, ale není tak veliká, aby se vyplatila grafická karta. Rychlost výpočtu na GPU je omezena nedostupností generátoru náhodných čísel rovnoměrného rozdělení přímo na GPU v knihovně `cltorch`. Výpočet je tak zásadně zpomalen kopírováním a čekáním na procesor.

8 Dodatky

8.1 Obsah CD

src zdrojové soubory trénovacího a vzorkovacího skriptu

doc text práce

models natrénované moduly pro JSB Chorales a Piano-MIDI.de

Literatura

Bayer, J. – Osendorfer, C. Learning Stochastic Recurrent Networks. *ArXiv e-prints*. November 2014.

Bengio, Y. – Simard, P. – Frasconi, P. Learning Long-term Dependencies with Gradient Descent is Difficult. *Trans. Neur. Netw.* March 1994, 5, 2, s. 157–166. ISSN 1045-9227. doi: 10.1109/72.279181. Dostupné z: http://dx.doi.org/10.1109/72.279181.

Bengio, Y. Learning Deep Architectures for AI. *Foundations and Trends® in Machine Learning*. 2009, 2, 1, s. 1–127. ISSN 1935-8237. doi: 10.1561/2200000006. Dostupné z: http://dx.doi.org/10.1561/2200000006.

Boulanger-Lewandowski, N. – Bengio, Y. – Vincent, P. Modeling Temporal Dependencies in High-Dimensional Sequences: Application to Polyphonic Music Generation and Transcription. *ArXiv e-prints*. June 2012.

Brychcín, T. Efektivní dekódování s N-gramovými jazykovými modely. Master's thesis, University of West Bohemia in Pilsen, 2010.

Chung, J. et al. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *CoRR*. 2014, abs/1412.3555. Dostupné z: http://arxiv.org/abs/1412.3555.

Felix, S. *DeepHear – Composing and harmonizing music with neural networks* [online]. 2015. Dostupné z: http://web.mit.edu/felixsun/www/neural-music.html.

Fischer, A. – Igel, C. *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications: 17th Iberoamerican Congress, CIARP 2012, Buenos Aires, Argentina, September 3-6, 2012. Proceedings*, An Introduction to Restricted Boltzmann Machines, s. 14–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. Dostupné z: http://dx.doi.org/10.1007/978-3-642-33275-3_2. ISBN 978-3-642-33275-3.

Gan, Z. et al. Deep Temporal Sigmoid Belief Networks for Sequence Modeling. In *NIPS*, 2015.

Goel, K. – Vohra, R. Learning Temporal Dependencies in Data Using a DBN-BLSTM. *CoRR*. 2014, abs/1412.6093. Dostupné z: http://arxiv.org/abs/1412.6093.

Goel, K. – Vohra, R. – Sahoo, J. K. Polyphonic Music Generation by Modeling Temporal Dependencies Using a RNN-DBN. *CoRR*. 2014, abs/1412.7927. Dostupné z: http://arxiv.org/abs/1412.7927.

Greff, K. et al. LSTM: A Search Space Odyssey. *CoRR*. 2015, abs/1503.04069. Dostupné z: http://arxiv.org/abs/1503.04069.

Hinton, G. E. *Neural Networks: Tricks of the Trade: Second Edition*, A Practical Guide to Training Restricted Boltzmann Machines, s. 599–619. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi: 10.1007/978-3-642-35289-8_32. Dostupné z: http://dx.doi.org/10.1007/978-3-642-35289-8_32. ISBN 978-3-642-35289-8.

Hinton, G. E. Training Products of Experts by Minimizing Contrastive Divergence. *Neural Computation*. August 2002, 14, 8, s. 1771–1800. ISSN 0899-7667. Dostupné z: http://dx.doi.org/10.1162/089976602760128018.

Hinton, G. E. – Osindero, S. – Teh, Y.-W. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*. July 2006, 18, 7, s. 1527–1554. ISSN 0899-7667. doi: 10.1162/neco.2006.18.7.1527. Dostupné z: http://dx.doi.org/10.1162/neco.2006.18.7.1527.

Hochreiter, S. – Schmidhuber, J. Long Short-Term Memory. *Neural Computation*. November 1997, 9, 8, s. 1735–1780. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. Dostupné z: http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Janeček, Z. Detekce a klasifikace základní frekvence zvukového záznamu za účelem nalezení hraného tónu. Bakalářská práce, University of West Bohemia in Pilsen, 2014.

Johnson, D. *Composing Music With Recurrent Neural Networks* [online]. 2015. Dostupné z: http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks.

Jurafsky, D. – Martin, J. Hidden Markov Models. *Speech and Language Processing*. 2014.

Larochelle, H. – Murray, I. The Neural Autoregressive Distribution Estimator. *JMLR: W&CP*. 2011, 15, s. 29–37.

LeCun, Y. et al. *Neural Networks: Tricks of the trade*, Efficient BackProp. Springer, 1998.

Léonard, N. et al. rnn : Recurrent Library for Torch. *CoRR*. 2015, abs/1511.07889. Dostupné z: http://arxiv.org/abs/1511.07889.

Lyu, Q. et al. Modelling High-Dimensional Sequences with LSTM-RTRBM: Application to Polyphonic Music Generation. In Yang, Q. – Wooldridge, M. (Ed.) *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, s. 4138–4139. AAAI Press, 2015. Dostupné z: http://ijcai.org/Abstract/15/582. ISBN 978-1-57735-738-4.

Mnih, A. – Gregor, K. Neural Variational Inference and Learning in Belief Networks. *CoRR*. 2014, abs/1402.0030. Dostupné z: http://arxiv.org/abs/1402.0030.

Murphy, K. P. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. ISBN 0262018020, 9780262018029.

Nair, V. – Hinton, G. E. 3D Object Recognition with Deep Belief Nets. In *Advances in Neural Information Processing Systems 22*, s. 1339–1347, 2009. Dostupné z: http://books.nips.cc/papers/files/nips22/NIPS2009_0807.pdf.

Nair, V. – Hinton, G. E. Rectified Linear Units Improve Restricted Boltzmann Machines. In Fürnkranz, J. – Joachims, T. (Ed.) *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, s. 807–814. Omnipress, 2010. Dostupné z: http://www.icml2010.org/papers/432.pdf.

Ngiam, J. et al. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, s. 265–272, 2011.

Nielsen, M. A. *Neural Networks and Deep Learning*. Determination Press, 2015. Dostupné z: http://neuralnetworksanddeeplearning.com.

Pascanu, R. et al. How to Construct Deep Recurrent Neural Networks. *CoRR*. 2013, abs/1312.6026. Dostupné z: http://arxiv.org/abs/1312.6026.

Schwehn, B. Using the Natural Gradient for training Restricted Boltzmann Machines. Master's thesis, University of Edinburgh, 2010.

Sigtia, S. – Benetos, E. – Dixon, S. An End-to-End Neural Network for Polyphonic Piano Music Transcription. *ArXiv e-prints*. August 2015.

Sturm, B. L. et al. Music transcription modelling and composition using deep learning. *ArXiv e-prints*. April 2016.

Sutskever, I. – Hinton, G. E. – Taylor, G. W. The Recurrent Temporal Restricted Boltzmann Machine. In Koller, D. et al. (Ed.) *Advances in Neural Information Processing Systems 21*. University of Toronto: Curran

Associates, Inc., 2009. s. 1601–1608. Dostupné z:
http://papers.nips.cc/paper/
3567-the-recurrent-temporal-restricted-boltzmann-machine.pdf.

Yosinski, J. – Lipson, H. Visually Debugging Restricted Boltzmann Machine Training with a 3D Example. In *Presented at Representation Learning Workshop, 29th International Conference on Machine Learning*, 2012.