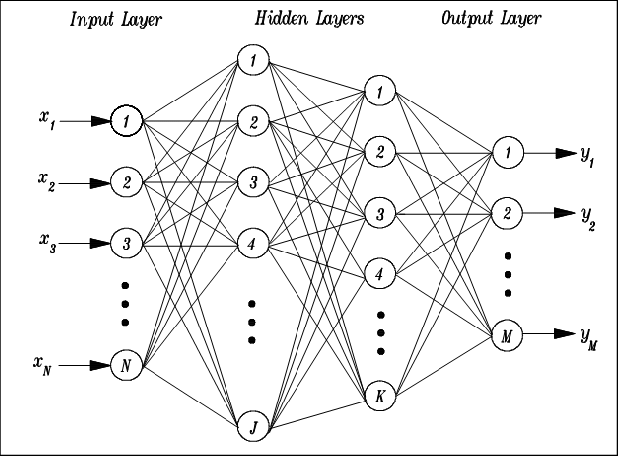
**4. Vícevrstvý perceptron**

**4.1 Inspirace a historický kontext**

*Vícevrstvý perceptron* (MLP – *multilayer perceptron*) je typ hluboké neuronové sítě, která se skládá z neuronů uspořádaných ve více vrstvách. Po představení MCP modelu neuronu a následné první implementaci Rosenblattova perceptronu v padesátých letech začal rychle upadat zájem o neuronové sítě. Nikdo totiž nedokázal vymyslet efektivní způsob, jakým trénovat více perceptronu ve více vrstvách. V roce 1986 však D.E.Rumelhart, G.E. Hinton a R.J. Wiliams ve své práci zpopularizovali algoritmus *zpětného šíření* (*backpropagation*)1 a obnovili tak ztracený zájem nejen většiny vědecké komunity, ale později i mnoha amatérů. V posledních několika letech došlo k obrovským pokrokům ve výzkumu hlubokých neuronových sítí obecně, hlavně díky posunu ve výrobě výkonných grafických karet, které jsou klíčové pro náročné výpočty. Se složitějšími architekturami tak dnes mohou pracovat i naprostí amatéři pouze za použití svého osobního počítače.

**4.2 Architektura MLP**



Obr. 4.1 Příkladné schéma architektury vícevrstvého perceptronu (MLP)

První dvě kapitoly této práce se týkaly jednovrstvých neuronových sítí (perceptron a ADALINE). MLP je na rozdíl od těchto algoritmů typově *hluboká neuronová síť* (DNN – *Deep Neural Network*), jelikož se skládá z více vrstev. Zároveň se jedná o *hustě propojenou* (*dense*) síť, takže každý neuron jedné vrstvy bude propojen se všemi neurony následné vrstvy. Takové hluboké sítě, které nejsou cyklické, označujeme jako *dopředné* (*feedforward*).

**4.2.1 Vrstvy**

Vrstvy neuronových sítí slouží k předávání a transformaci vstupních dat pomocí aktivačních funkcí. Existuje mnoho způsobů, jakými se dají tyto vrstvy nastavit podle zamýšleného využití (například rozpoznávání vizuálních nebo zvukových dat, klasifikace, či generování nových dat). Vrstvou se rozumí soubor neuronů se stejnými aktivačními funkcemi, které jsou zároveň propojeny s neurony dalších vrstev (v případě husté sítě plně). MLP obsahuje (ostatně jako všechny hluboké sítě) *vstupní vrstvu* (*input layer –* vkládáme do ní vlastní zadání příkladu), *výstupní vrstvu* (*output layer –* je v ní výsledek výpočtu sítě) a jednu nebo více *skrytých vrstev* (*hidden layers*). Neuronová síť je označována za hlubokou právě tehdy, když má alespoň dvě skryté vrstvy. Vstupní vrstva slouží k přebrání vstupních dat, tedy příznaků, zatímco výstupní vrstva naopak transformuje vstupní data na požadované hodnoty (například můžeme chtít jako výstup pravděpodobnosti cílových tříd). Skryté vrstvy jsou přidávány právě kvůli nelinearitě a jejich přesnou funkci vysvětlím později. Na obrázku 4.1 je vidět schéma MLP se dvěma skrytými vrstvami. Opět zdůrazňuji, že se jedná o síť hustou, takže každý neuron jedné vrstvy je propojen se všemi neurony druhé vrstvy.

**4.2.2 Neurony a synapse**

Neurony v jednotlivých vrstvách MLP fungují naprosto stejně jako perceptron (odsud název vícevrstvý perceptron). Vstupní data jsou sítí předávána jako příznaky daného příkladu, takže počet neuronů ve vstupní vrstvě odpovídá počtu příznaků. Na obrázku 4.2 je vidět hustá síť s jednou skrytou vrstvou (vstupní vrstva je plně propojena se skrytou vrstvou, ta je zase plně propojena s vrstvou výstupní). Počet neuronů ve skrytých vrstvách je volitelný, počet neuronů ve výstupní vrstvě je obvykle určen počtem tříd (pokud pracujeme s více než dvěma třídami, jedná se o multinomiální klasifikaci a predikovaná třída je určena neuronem s nejvyšší hodnotou). Synapse, kterými jsou jednotlivé neurony propojeny, představují váhy, které se budeme Diagram

Description automatically generatedsnažit optimalizovat během učení sítě (stejně jako u ADALINE a perceptronu).

Obr. 4.2 Hustě propojená síť s jednou skrytou vrstvou

**4.3 Dopředené šíření**

Proces trénování sítě ve smyslu předávání vstupních dat je obvykle označován jako *dopředné šíření* (*forward propagation*). Jedná se postupné transformování dat pomocí jednotlivých vrstev. Pro zjednodušení budu celý tento proces vysvětlovat na MLP s jednou skrytou vrstvou, avšak postup je v tomto případě obecný a lze ho aplikovat i na hluboké sítě tohoto typu.

**4.3.1 Aktivace skryté vrstvy**

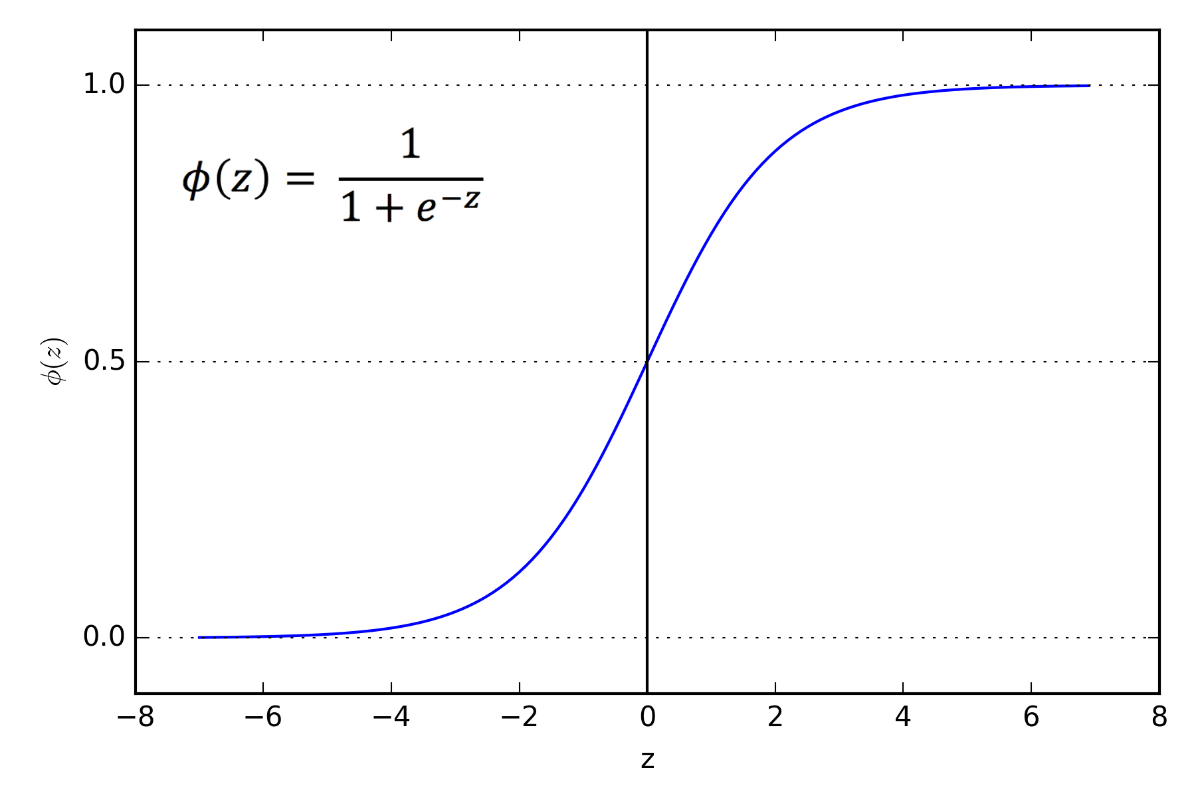
Hodnoty neuronů **A(in)** ve vstupní vrstvě jsou identické s hodnotami příznaků **X(i)**:

Jedinou výjimkou je první neuron , který obvykle představuje bias. Počet neuronů ve vstupní vrstvě tak odpovídá počtu příznaků **.** Prvním krokem při dopředném šíření je výpočet hodnot neuronů ve skryté vrstvě. Stejně jako u perceptronu musíme nejdříve vypočítat lineární vstup daného neuronu, a poté na něj aplikovat aktivační funkci. Stejně jako vstupní vrstva, skrytá též obsahuje jednotku biasu . Jelikož se jedná o síť hustou, k výpočtu lineárního vstupu libovolného neuronu ve skryté vrstvě sečteme násobky všech neuronů vstupní vrstvy (včetně biasu s násobkem 1) a jejich příslušných vah **)** (příslušnou váhou se myslí konkrétní synapse mezi oběma neurony):

nebo:

Horním indexem *T* značíme opět operaci transpozice, tedy matici převrácenou kolem diagonály.

V dalším kroku aplikujeme na lineární vstup aktivační funkci . Často používanou aktivační funkcí v případě MLP je například logistická aktivační funkce (sigmoida), která transformuje vstup na hodnotu od 0 do 1:

****

Obr. 4.3 Graf logistické aktivační funkce (sigmoidy)

Hodnotu libovolného neuronu skryté vrstvy můžeme tedy zapsat jako:

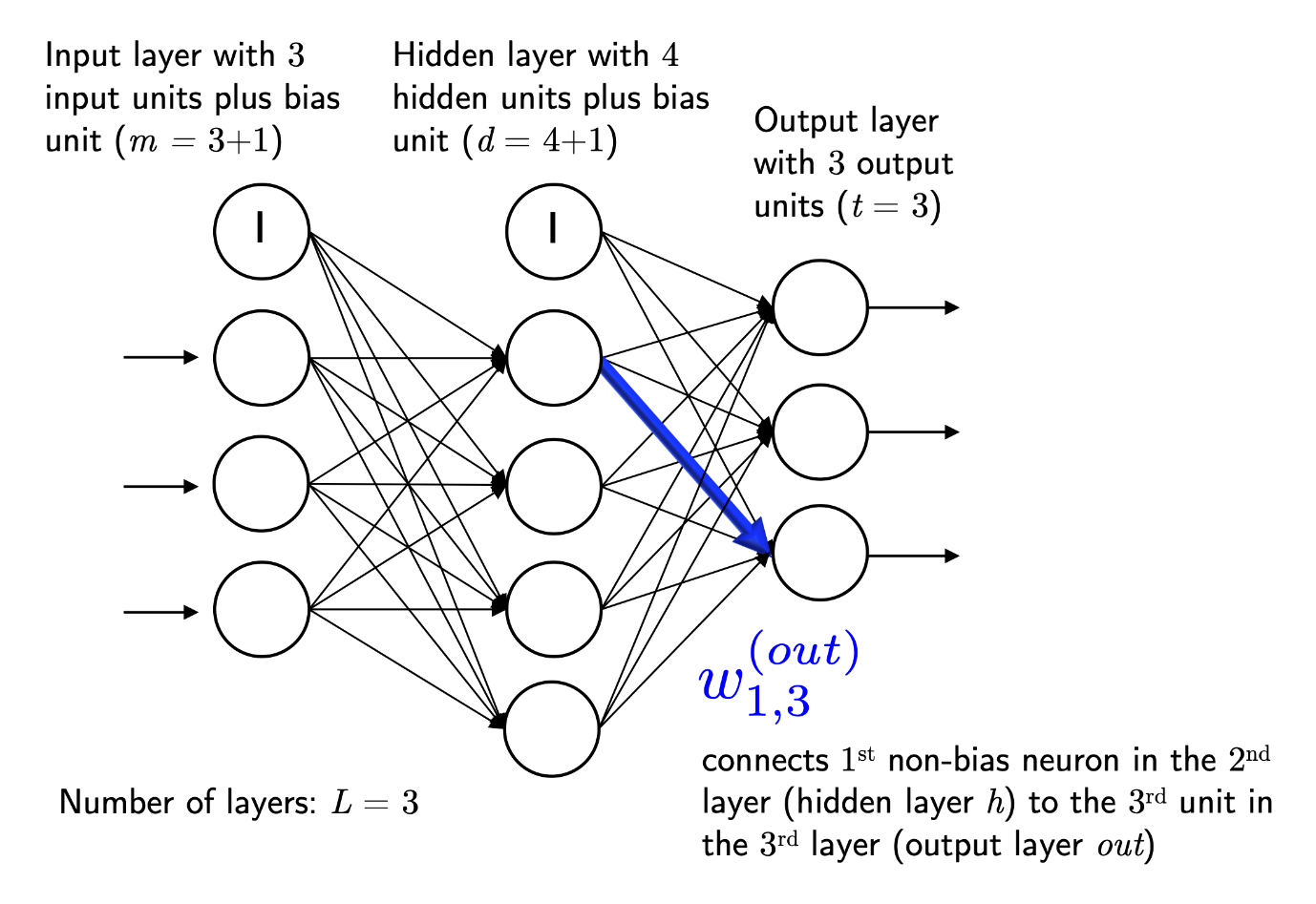
**4.3.2 Aktivace výstupní vrstvy**

Výpočet hodnot neuronů výstupní vrstvy je naprosto stejný jako u skryté vrstvy. Výstupní vrstva samozřejmě na rozdíl od vstupní a skryté neobsahuje bias, ale pouze výstupní neurony Opět připomínám, že MLP je hustá síť, takže všechny neurony výstupní vrstvy jsou propojeny se všemi neurony předchozí skryté vrstvy. Hodnota libovolného výstupního neuronu je tedy opět:

Je důležité zvolit správnou aktivační funkci . Její výběr je určen tvarem výstupu, který od sítě požadujeme. Můžeme například opět použít logistickou aktivační funkci – výstupní neuron s nejvyšší hodnotou bude poté představovat predikovanou třídu. Pokud bychom však chtěli vidět procentní pravděpodobnosti jednotlivých neuronů **A(out)** výstupní vrstvy (tento přístup je obvykle označován jako *pravděpodobnostní hluboké učení* – *probabilistic deep learning*), tedy pravděpodobnost, s kterou daný příklad spadá do konkrétní kategorie, bylo by výhodnější použít funkci *softmax*. Ta sice stejně jako sigmoida transformuje vstup na hodnotu od 0 do 1, ale zároveň bere do úvahy všechny výstupní neurony (v případě výstupní vrstvy se tak jedná o ostatní třídy). Součet hodnot všech neuronů ve vrstvě s aktivační funkcí softmax bude 1, takže použijeme-li tuto funkci ve výstupní vrstvě místo sigmoidy, můžeme hodnoty neuronů interpretovat jako pravděpodobnosti:

Proměnná ***K***ve jmenovateli vzorce představuje počet tříd. Celý jmenovatel je tzv. *normovací konstanta* (*normalization term*), který je zodpovědný za onu pravděpodobnostní distribuci (součet výsledných hodnot vyjde jako 1). V případě určování z více než 2 tříd se s použitím softmax jedná o *mnohonásobnou logistickou regresi* (*multinomial logistic regression*).9

Poté, co data projdou všemi vrstvami sítě, je potřeba vypočítat ztrátové skóre a následně posílit nebo oslabit synapse mezi neurony (optimalizovat váhové koeficienty).



Obr. 4.4 Schéma MLP s jednou skrytou vrstvou se zvýrazněnými biasy

**4.4 Ztrátová funkce – křížová entropie**

Jak jsem již vysvětloval ve 2. kapitole, jedním z hlavních pilířů učení neuronových sítí je ztrátová funkce. Ta slouží k výpočtu ztrátového skóre, které se následně snažíme minimalizovat různými metodami optimalizace pro co nejlepší výkon sítě – v případě ADALINE se jednalo o jednoduchou kvadratickou sumu odchylek. Samozřejmě nejsme vázání pouze jednou ztrátovou funkcí na každý typ sítě, v praxi se s výběrem hodně experimentuje. Při MLP můžeme použít například ztrátovou funkci (v teorii informace je tato funkce označována je křížová entropie **–** „cross entropy“**):**

Výraz představuje logistickou aktivaci *i*-tého příkladu1, o které jsem mluvil v podkapitole 4.3.1. Tuto funkci ještě musíme upravit tak, aby brala v úvahu všechny výstupní aktivační jednotky **t** naší sítě:

**4.4.1 Regularizace ztrátové funkce**

Abychom snížili riziko přetrénování sítě, můžeme na ztrátovou funkci aplikovat takzvanou L2 regularizaci:

Neboli:

Logistická ztrátová funkce s L2 regularizací bude tedy vypadat takto:

Neboli:

**+**

Zjednodušeně se jedná pouze o přičtení součtu všech váhových koeficientů vrstvy ***l***. Použité L2 regularizace není zásadní, avšak výkon modelu se může zlepšit (jak jsem již zmínil, L2 značně snižuje šanci na přetrénování).

**4.5 Zpětné šíření chyby**

*Algoritmus zpětného šíření chyby* (*backpropagation algorithm*) je velmi mocný nástroj pro optimalizaci váhových koeficientů mnoha typů neuronových sítí.

**4.6 Souhrn**

MLP (vícevrtsvý perceptron) je umělá neuronová síť schopná provádět multionomiální klasifikaci. Skládá se z vstupní vrstvy (příznaky příkladu), jedné nebo více skrytých vrstev (kvůli přidání nelinearity) a výstupní vrstvy (ta obsahuje výsledné hodnoty). Každá vrstva (kromě vstupní) aplikuje na hodnoty svých neuronů (lineární vstupy) jisté aktivační funkce. Jelikož MLP je hustě propojená síť, každý neuron jedné vrstvy je propojen s každým neuronem další vrstvy synapsemi, které představují váhy. Všechny vrstvy kromě výstupní obsahují také bias (synapse s biasem mají hodnotu 1). Při trénování MLP jsou nejdříve náhodně určeny hodnoty všech vah. Poté je provedeno dopředné šíření (jedná se dopřednou síť), při kterém se vypočítají hodnot všech neuronů a aplikují se na ně aktivační funkce. Následně je určena hodnota ztrátové funkce a provádí se zpětná propagace chyby, při které jsou aktualizovány váhové koeficienty na základě gradientu ztrátové funkce. Trénování opět probíhá v epochách. Na konci toho procesu je model schopen provádět multinomiální (nebo binární) klasifikaci.

**4.7 Implementace**

Implementace MLP v Pythonu bude nejsložitější a výpočetně nejnáročnější ze všech dosavadních příkladů. To nám však poskytuje dobrou příležitost k demonstraci síly a efektivity populárního frameworku *Keras* v rámci knihovny *TensorFlow* (viz kapitola 1). Jak jsem již vysvětloval v úvodu této práce, jedná se o knihovnu, která byla vyvinuta společností Google specificky pro účely strojového učení. Uživatelé v ní dokážou jednoduše implementovat obrovské množství běžných algoritmů i velmi složitých neuronových sítí, jejichž trénování probíhá relativně rychle díky skvělé optimalizaci výpočtů, kterou TensorFlow zprostředkovává. Framework Keras poté slouží jako uživatelsky přívětivější nadstavba této knihovny. V následujícím příkladu tedy nejdříve představím implementaci MLP v prostém Pythonu, a poté pro porovnání představím ten stejný algoritmus napsaný ve frameworku Keras.

**4.7.1 MNIST**

Další výhodou MLP oproti předchozím příkladům je kromě lepší optimalizace vah schopnost multinomiální klasifikace. Pro demonstraci rozdělování dat do více tříd budu proto využívat databázi MNIST namísto Iris. Jedná se velmi oblíbenou databázi obsahující 60 000 trénovacích a 10 000 testovacích černobílých obrázků rukou psaných číslic s rozměry 28x28 pixelů. V následujícím příkladu budu pracovat celkem s 10 třídami (číslice 0-9), takže se nejedná o algoritmus binární klasifikace, jako tomu bylo u Perceptronu a ADALINE, ale jde o již zmíněnou multinomiální klasifikaci. MLP se tedy bude snažit naučit, jak vypadají jednotlivé číslice, aby je následně dokázal přečíst. MNIST lze v Pythonu načíst mnoha způsoby, já v tomto příkladu pracuji se soubory staženými z této stránky: <http://yann.lecun.com/exdb/mnist/>

O zpracování stažených souborů tak, aby se poté daly jednoduše načítat z jediného souboru se stará následující kód:



Kód 7 – load\_mnist.py

**4.7.2 MLP v prostém Pythonu**

Implementace MLP v prostém Pythonu je o poznání složitější než u předchozích příkladů (hlavně kvůli zpětnému šíření chyby), proto se pokusím vysvětlit hlavně klíčové části programu, aby tato kapitola nebyla zbytečně příliš dlouhá. Neuronová síť reprezentuje třída *MLP\_NeuralNetwork()*:



Kód 5 – MLP.py

Funkce parametrů třídy *MLP\_NeuralNetwork()* jsou z kódu celkem jasné. Matoucí by mohl být parametr *minibatch\_size*, který určuje, kolik trénovacích příkladů bude obsahovat každá dávka, na kterých bude jednotlivě probíhat gradientní optimalizace. Tento přístup je výpočetně efektivnější než práce s celým souborem najednou (viz kapitola 3). O trénování algoritmu se opět stará (tentokrát poměrně složitá) metoda *fit(X\_train, y\_train, X\_valid, y\_valid)*:



Kód 5 – MLP.py

Tato metoda je dále rozdělena na celkem tři hlavní části. Nejdříve probíhá náhodná inicializace vah a výpočet aktivací jednotlivých neuronů (skrytá i výstupní vrstva používají jako aktivační funkci sigmoidu):



Kód 5 – MLP.py

Jak je z kódu vidět, hodnoty všech biasů *self.b\_h* a *self.b\_out* jsou na počátku trénování určeny jako 0. Dále je deklarován atribut *self.eval\_*, který bude sloužit k ukládání evaluačních dat z každé epochy (ta nám budou sloužit ke sledování pokroku v učení algoritmu). Tento kód zároveň odkazuje na metodu *\_onehot(y, n\_classes)*:



Kód 5 – MLP.py

Tato metoda převádí vektor cílových tříd na matici s one-hot reprezentacemi těchto tříd. One-hot je v oblasti strojového učení běžně používané kódování, které každou cílovou třídu převede na vektor nul, kde jediná jednička představuje onu třídu. Například cílové třídy 0-5 by ve one-hot kódování vypadaly takto:

Tímto způsobem se dá celý vektor cílových tříd trénovací datové množiny zapsat jako matice 10x60000. Druhou částí metody *fit(X\_train, y\_train, X\_valid, y\_valid)* je výpočet zpětného šíření chyby:



Kód 5 – MLP.py

Poslední částí je evaluace modelu. V této částí algoritmus vypočítá momentální ztrátové skóre a přesnost predikcí na trénovací a validační datové množině. Validační příklady slouží k objektivnímu posouzení přesnosti predikce na nových (předtím neviděných) datech během trénování (testovací příklady se používají až s natrénovaným modelem):

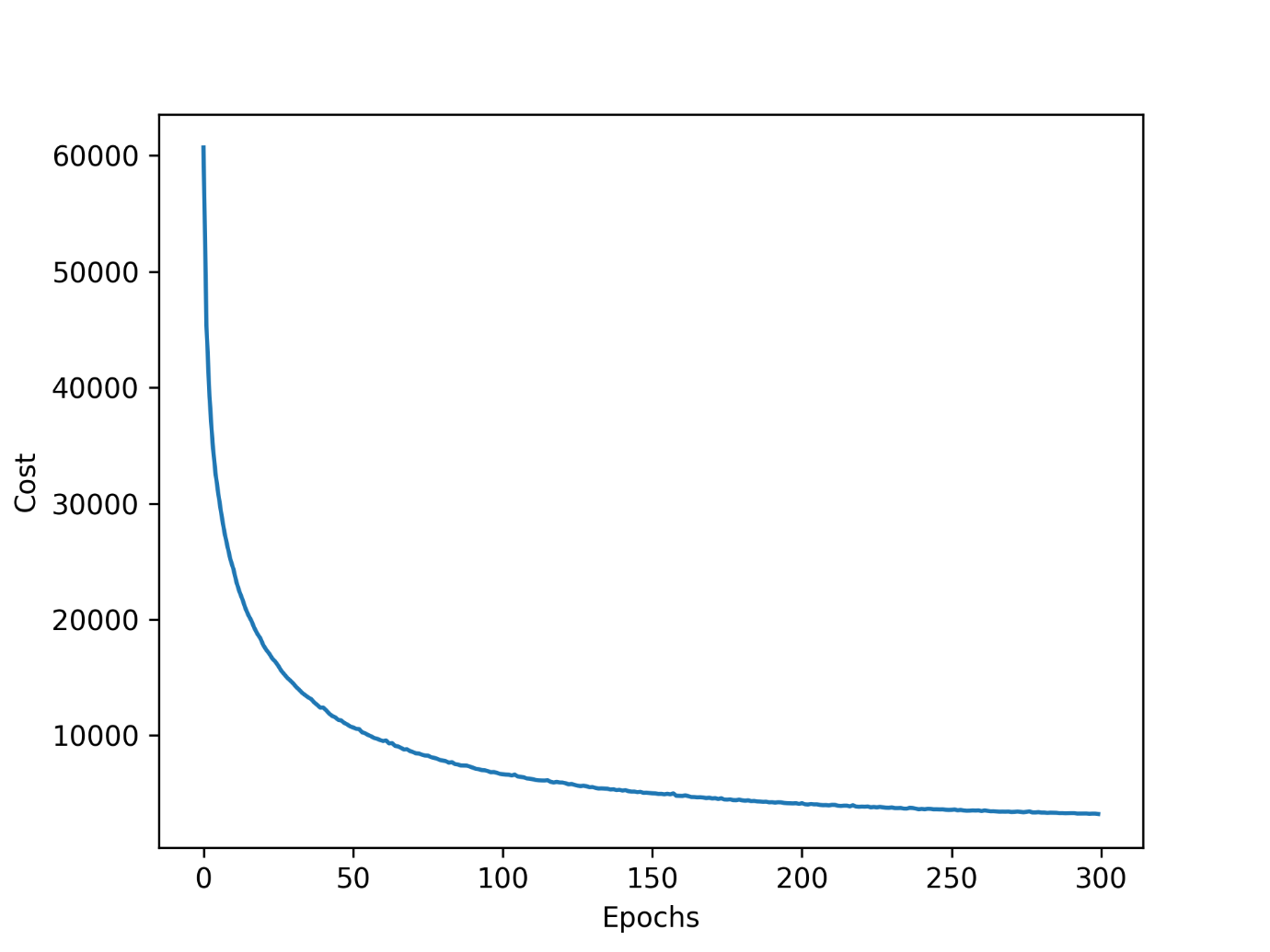


Kód 5 – MLP.py

Nebudu zde popisovat pomocné metody pro výpočet dopředného šíření, ztrátové a logistické aktivační funkce, jelikož se jedná pouze o přepsání uvedených rovnic do Pythonu. Veškerý kód je uveden v příloze s podrobnými komentáři. K natrénování sítě jsem použil následující kus kódu:

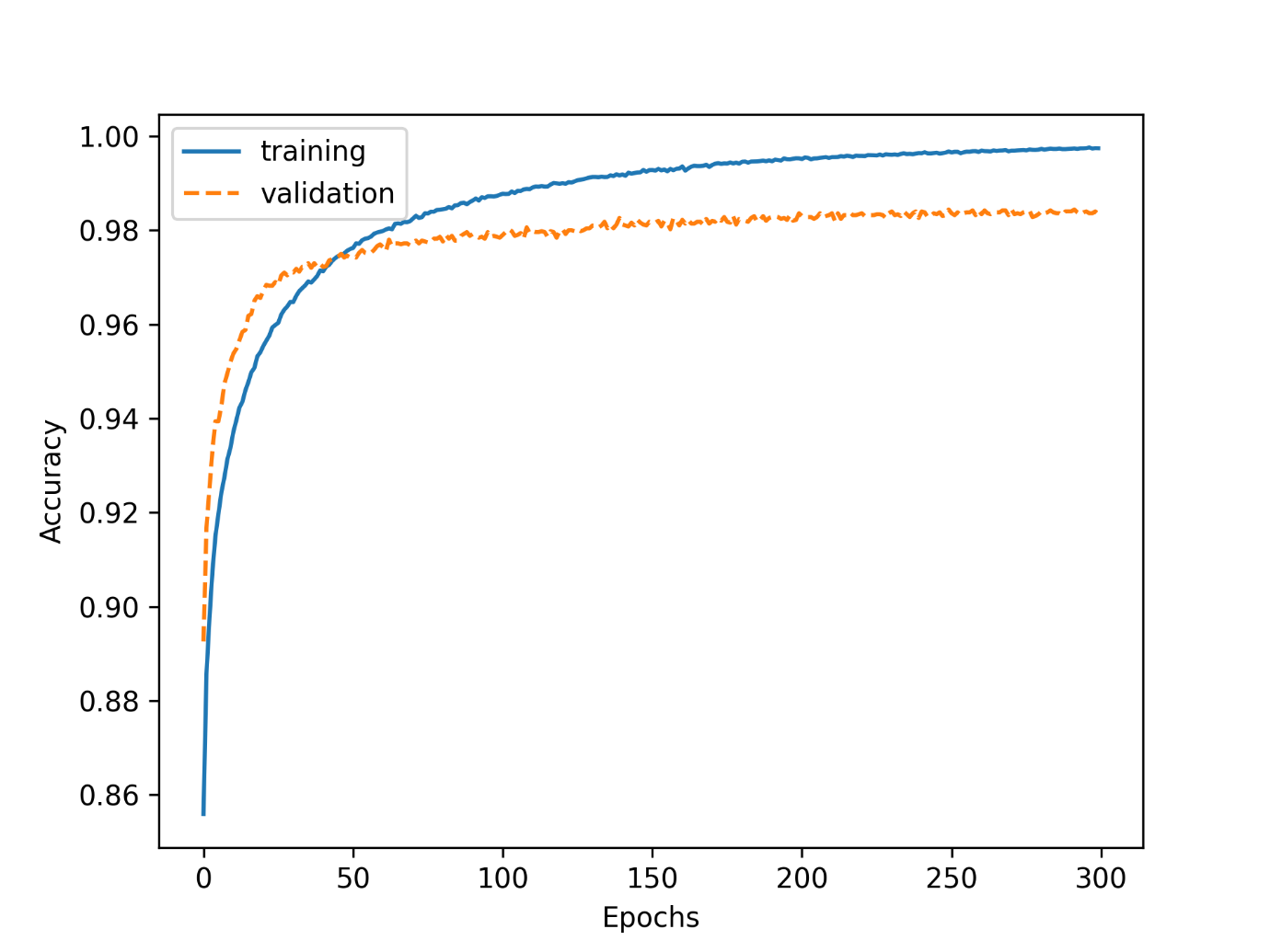


Kód 6 – mlp\_mnist.py

Následující graf ukazuje, jakým způsobem se během trénování snižovalo průměrné ztrátové skóre funkce křížové entropie během 300 epoch při rychlosti učení 0,0005 s regularizací L2 o hodnotě 0,01, velikostí dávky 100 příkladů (s povoleným mícháním trénovacích příkladů) a 150 neuronech ve skryté vrstvě:

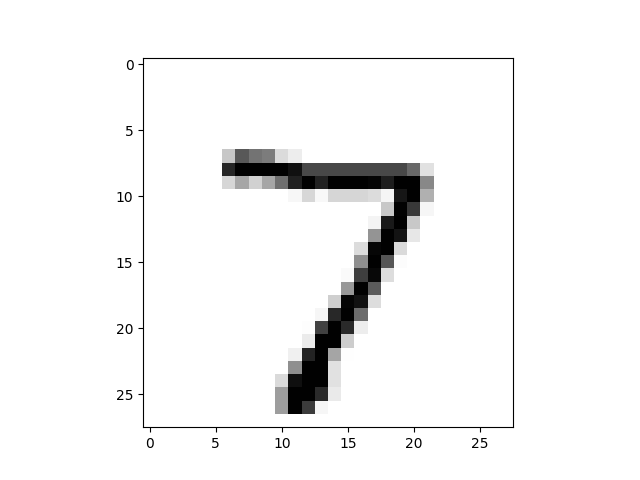
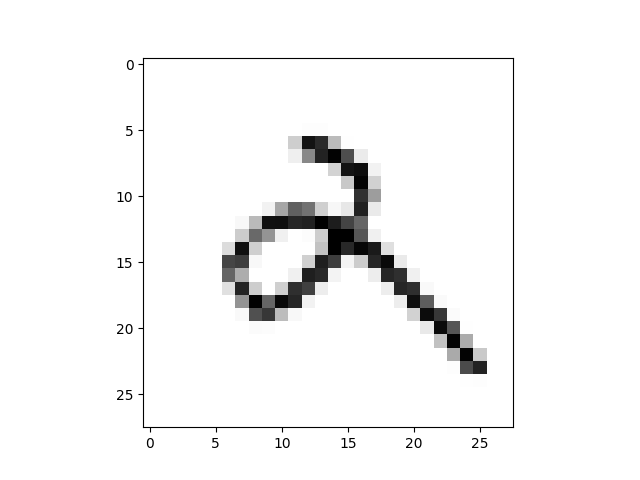
Obr. 4.5 Graf zobrazující průměrné hodnoty křížové entropie během 300 epoch

Na grafu 4.6 zase můžeme vidět rozdíl mezi přesnostmi predikcí na trénovacím a validačním datovém souboru během učení (právě tento rozdíl demonstruje, proč je dobrou praxí používat validační množinu k objektivnímu zhodnocení pokroku sítě během jejího trénování):



Obr. 4.6 Graf znázorňující rozdíl mezi přesnostmi predikcí na trénovací a validační datové množině během trénování

Pro ukázku fungování natrénovaného MLP můžeme na obrázku 4.7 vidět číslo 7, které natrénovaný model určil správně. Na obrázku 4.8 se zase nachází číslo 2, které neuronová síť chybně určila jako číslo 9. Jedná se o číslice z testovací části databáze MNIST:



Obr 4.8 Číslice 2 chybně určená jako číslo 9

Obr 4.7 Správně rozpoznaná Číslice 7

Přesnost natrénovaného MLP byla 98,06 %.

**4.7.3 MLP v TensorFlow Keras**

V minulé kapitole jsme si ukázali implementaci MLP v prostém Pythonu. Nyní chci demonstrovat sílu a efektivitu frameworku Keras v rámci knihovny TensorFlow. Nejdříve musíme importovat potřebné balíčky z Kerasu:



Kód 8 – Keras\_MLP.py

Jak můžeme vidět, framework *keras* je importován z knihovny *tensorflow*. Na prvním řádku je importována databáze MNIST přímo z frameworku Keras, což je daleko efektivnější způsob načítaní tohoto souboru, než jaký jsem použil při implementaci MLP v prostém Pythonu. Dále je z Kerasu importován *Sequential* model (jedná se lineární typ modelu) a *Dense* vrstva (hustá – plně propojená). Funkce *to\_categorical* slouží k zakódování cílových tříd podle one-hot.

Následující část kódu se věnuje přípravě datového souboru:



Kód 8 – Keras\_MLP.py

Poté zbývá pouze sestavit model a spustit trénování:



Kód 8 – Keras\_MLP.py

Toto je kompletní kód potřebný k implementaci MLP se stejnými parametry a konfigurací (kromě aktivační funkce výstupní vrstvy – zde jsem použil softmax namísto sigmoidy) jako při implementaci v prostém Pythonu. Je nepochybně kratší, jednodušší na čtení, výpočetně efektivnější a snadno upravovatelný – to vše díky knihovně TensorFlow a frameworku Keras. Přesnost tohoto MLP byla 95,5 % a ztrátové skóre na konci trénování 0,15.