DOKUMENTÁCIA

Klasifikácia MNIST a Backpropagation algoritmus

Zadanie 3a

Johanna Tilešová | xtilesova@stuba.sk

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačnej bezpečnosti

9. december 2024

Obsah

KLASIFIKÁCIA MNIST **3**

Zadanie 3
Cieľ zadania 3
Očakávaný výstup 3
Riešenie 4
Implementácia dát *4*
Definovanie modelu *5*
Definovanie parametrov *6*
Funkcia na trénovanie *7*
Porovnanie optimalizačných algoritmov 9
SGD – Stochastic gradient descent *9*
SGD s momentom *10*
ADAM – Adaptive Moment Estimation *11*
Vyhodnotenie MNIST klasifikácie 12

BACKPROPAGATION ALGORITMUS **13**

Zadanie 13
Cieľ zadania 13
Očakávaný výstup 13
Riešenie 14
Reprezentácia dát *14*
Definovanie siete *16*
Forward a backward metóda *17*
Funkcia na trénovanie *18*
Vyhodnotenie algoritmu Backpropagation 19
Sigmoid *19*
Tanh *19*
ReLU *20*

Klasifikácia MNIST

Zadanie

Úlohou bolo vytvoriť neurónovú sieť, ktorá bude schopná klasifikovať ručne písané číslice z dátového súboru MNIST. Dáta set je zostavený zo 70 000 obrázkov, z toho 60 000 obrázkov je určených na učenie a 10 000 obrázkov je na testovanie. Každý obrázok reprezentuje jednu číslicu od 0 po 9, ktorá je znázornená pomocou odtieňov šedej farby a vo veľkosti 28x28.

Cieľ zadania

Cieľom úlohy bolo natrénovať danú neurónovú sieť pomocou troch algoritmov – SGD *(Stochastic gradient descent)*, SGD s momentom a ADAM *(Adaptive Moment Estimation)* a následne pomocou grafov odmerať tréningovú a testovaciu chybu a presnosť modelu.

Očakávaný výstup

Na výstupe sa očakáva natrénovanie neurónovej siete s výslednou presnosťou nad 97% na testovacej množine a následné vyhodnotenie a porovnanie algoritmov.

Riešenie

Implementácia dát

```
transform = transforms.Compose([transforms.ToTensor()])

train_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=True,
    download=True,
    transform=transform)

test_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=False,
    download=True,
    transform=transform)

loaders = {
    'train': DataLoader(train_dataset, batch_size=64, shuffle=True),
    'test': DataLoader(test_dataset, batch_size=64, shuffle=False),
}
```

Na začiatku sa pomocou funkcie `transforms.Compose()` pre každý obrázok v datasete skonvertujú hodnoty pixelov z rozsahu 0 až 255 na hodnoty medzi 0 až 1.

Dáta sú implementované pomocou načítaného datasetu MNIST, ktorý je rozdelený na dve množiny, a to množinu na trénovanie, ktorá obsahuje 60 tisíc obrázkov a množinu na testovanie, ktorá pozostáva z 10 tisíc obrázkov.

Oba datasety sú následne rozdelené do loaderov, ktoré ich rozdelia na menšie časti, v tomto prípade do 64 obrázkov (`batch_size=64`), pre jednoduchšie spracovanie dát počas trénovania a testovania. Dáta v trénovanej množine sa premiešajú, aby nedošlo k pretrénovaniu a dáta v testovacej množine sa nepotrebujú miešať, aby sa dosiahli konzistentné výsledky.

Definovanie modelu

```
class MNISTModel(nn.Module):  2 usages
    def __init__(self):
        super(MNISTModel, self).__init__()
        self.fc1 = nn.Linear( in_features: 784,  out_features: 128)
        self.fc2 = nn.Linear( in_features: 128,  out_features: 64)
        self.fc3 = nn.Linear( in_features: 64,  out_features: 10)
        self.relu = nn.ReLU()
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        x = x.view(-1, 784)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.softmax(self.fc3(x))
        return x
```

Definujeme si model MNISTModel, ktorý dedí všetky vlastnosti z triedy nn.Module z knižnice PyTorch. Inicializácia modelu pozostáva z 3 plne prepojených vrstiev *(fully connected layer)*, kedy každá má iný počet výstupov a výstupov.

Na prvej vrstve vstupuje 784 vstupov, čo predstavuje celý jeden obrázok 28*28 pixelov a počet výstupov je 128. Do druhej vrstvy vstupuje 128 vstupov a vystupuje z nej 64 výstupov. Tretiu vrstvu reprezentuje 64 vstupov a 10 výstupov, čo sú vlastne výsledné obrázky s číslami od 0 po 9.

Po každej vrstve sa aplikuje aktivačná funkcia ReLU (Rectified Linear Unit), ktorá nastavuje záporné hodnoty na 0. Na konci sa používa Softmax, ktorý prevedie výstupy na pravdepodobnosti pre jednotlivé triedy.

Forward metóda

Vstupné dáta 28*28 pixelov sa preformátuje na jednorozmerné pole s veľkosťou 784 pixelov. Následne dáta prechádzajú prvou vrstvou a aktivačnou funkciou ReLU, potom výsledok prvej vrstvy prechádza do druhej vrstvy a tiež sa aplikuje aktivačná funkcia ReLU. Výstup druhej vrstvy prechádza treťou vrstvou, kde sa aplikuje Softmax. Funkcia vráti hodnotu, ktorá obsahuje pravdepodobnosti pre jednotlivé číslice od 0 po 9.

Definovanie parametrov

```python
# 3. Definovanie parametrov
learning_rate = 0.0001  # Rýchlosť učenia
epochs = 50 # Počet epoch

# 4. Loss funkcia a optimalizačné algoritmy
criterion = nn.CrossEntropyLoss()

optimizers = {
    'SGD': lambda model: optim.SGD(model.parameters(), lr=learning_rate),
    'SGD with momentum': lambda model: optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9),
    'Adam': lambda model: optim.Adam(model.parameters(), lr=learning_rate),
}
```

Na trénovanie siete bola použitá rýchlosť učenia 0,0001 pre stabilnejšie učenie a bola učená na 50 epochách, takže prešla 50x celým datasetom.

Výpočet stratovej funkcie pomocou `criterion = nn.CrossEntropyLoss()`. Funkcia porovnáva výstupy so skutočnými triedami.

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\ \text{size}}} y_i \cdot \log \hat{y}_i$$

Funkcia na trénovanie

```python
# Funkcia na trénovanie modelu
def train_model(optimizer_name):  1 usage
    model = MNISTModel()  # znova vytvoríme model, aby sme mohli trénovať odznova pre každý optimizer
    optimizer = optimizers[optimizer_name](model)
    train_losses = []
    test_losses = []
    train_accuracies = []
    test_accuracies = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        correct_train = 0
        total_train = 0
        for images, labels in loaders['train']:
            optimizer.zero_grad() # Vynulovanie gradientov
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total_train += labels.size(0)
            correct_train += (predicted == labels).sum().item()
        train_losses.append(running_loss / len(loaders['train']))
        train_accuracies.append(100 * correct_train / total_train)

        model.eval()
        test_loss = 0.0
        correct_test = 0
        total_test = 0
        with torch.no_grad():
            for images, labels in loaders['test']:
                outputs = model(images)
                loss = criterion(outputs, labels)
                test_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                total_test += labels.size(0)
                correct_test += (predicted == labels).sum().item()
        test_losses.append(test_loss / len(loaders['test']))
        test_accuracies.append(100 * correct_test / total_test)

        print(f'Epoch [{epoch+1}/{epochs}], Training Loss: {train_losses[-1]:.4f}, '
              f'Testing Loss: {test_losses[-1]:.4f}, '
              f'Training Accuracy: {train_accuracies[-1]:.2f}%, '
              f'Testing Accuracy: {test_accuracies[-1]:.2f}%')

    return train_losses, test_losses, train_accuracies, test_accuracies, model
```

Táto funkcia slúži na vytrénovanie neurónovej siete pomocou rôznych optimalizačných algoritmov. Na začiatku sa vždy pre každý algoritmus inicializuje nový model, aby sa trénovanie vždy začínalo s pôvodným nastavením váh.

Trénovanie modelu

Model spracováva celý trénovací dataset postupne po menších častiach (batchoch), aby bolo trénovanie efektívne a nezaberalo veľa pamäte. Po každom batchi sa spočíta stratová funkcia, ktorá meria chyby modelu. Pomocou gradientov a optimalizačného algoritmu sa aktualizujú váhy modelu, aby sa minimalizovala strata.

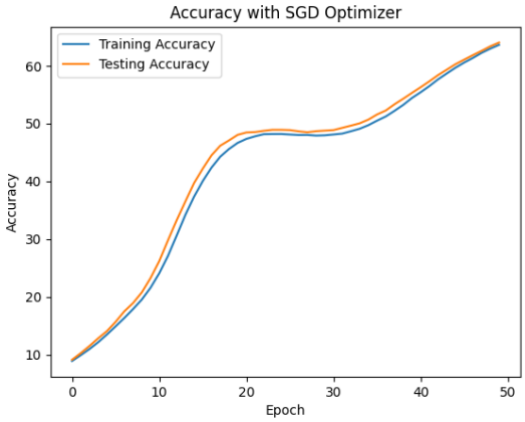
Testovanie modelu

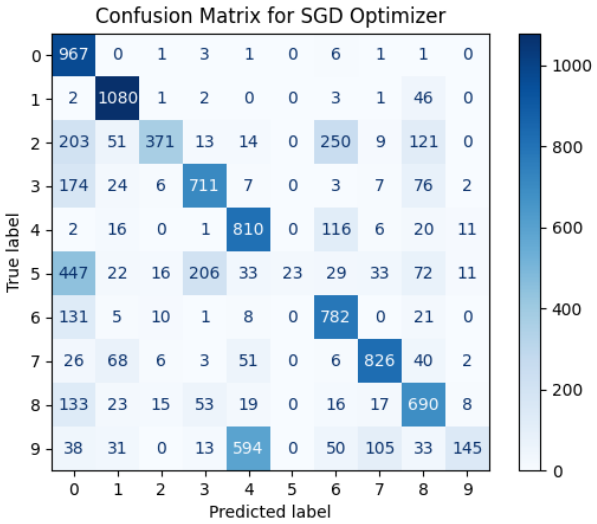
Testovanie modelu prebieha na nezávislej testovacej množine, ktorá nebola použitá pri trénovaní. Počas testovania sa váhy modelu nemenia. Jej cieľom je vyhodnotiť ako dobre model rozpoznáva neznáme dáta. Výstupom testovania je priemerná testovacia strata a presnosť modelu.

Porovnanie optimalizačných algoritmov

SGD – Stochastic gradient descent

SGD je variant gradientného zostupu, ktorý aktualizuje váhy modelu na základe chýb vypočítaných z jednej náhodne vybranej vzorky alebo malého batchu dát. SGD nepočíta gradient na celom trénovanom súbore, ale znižuje čas výpočtu a umožňuje častejšie aktualizácie. Často môže byť nestabilný, pretože aktualizácie závisia od náhodne vybranej vzorky.

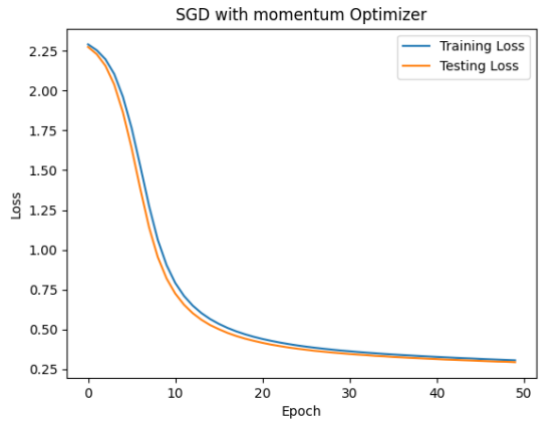


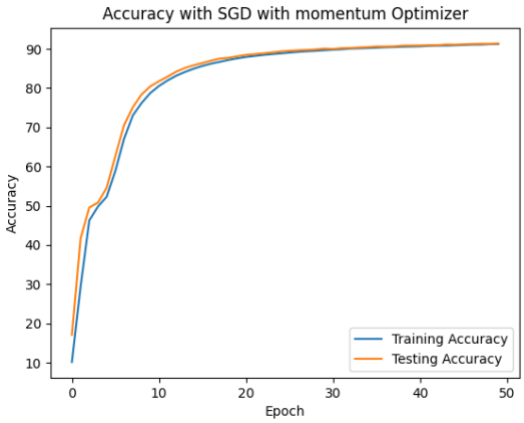


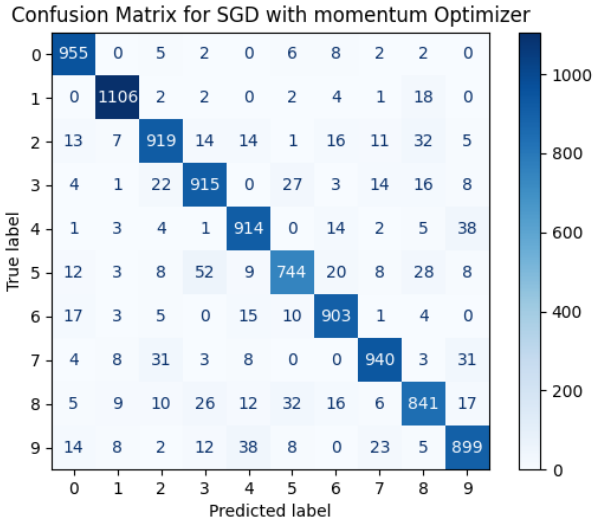
Z grafov vidíme, že optimalizačný algoritmus SGD nie je úplné presný a má vysoké rozdiely pri rozpoznávaní obrázkov. Jeho presnosť po 50 epochách nepresiahne viac ako 63,65% pri testovaní. Podľa confusion matrix môžeme tiež vidieť, že častokrát došlo k pomýleniu a nesprávnemu rozoznaniu čísla.

SGD s momentom

Ide o rozšíreny SGD algoritmus, ktorý využíva momentum k aktualizáciám váh. Sleduje predchádzajúce aktualizácie a kombinuje ich s aktuálnym gradientom. Nevýhodou môže byť, ak je momentum príliš veľké, tak model môže preskočiť optimálne riešenie.



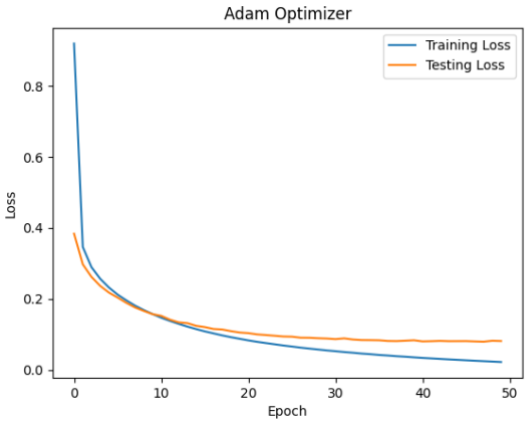


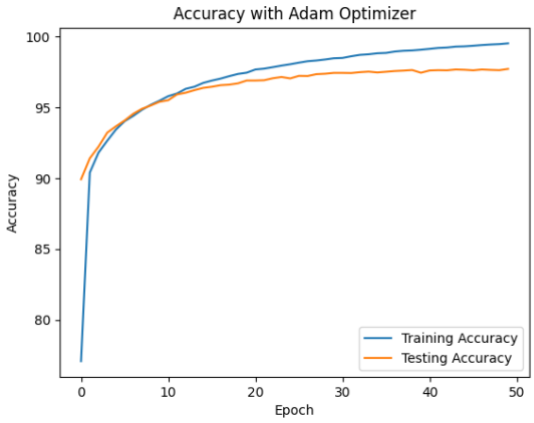


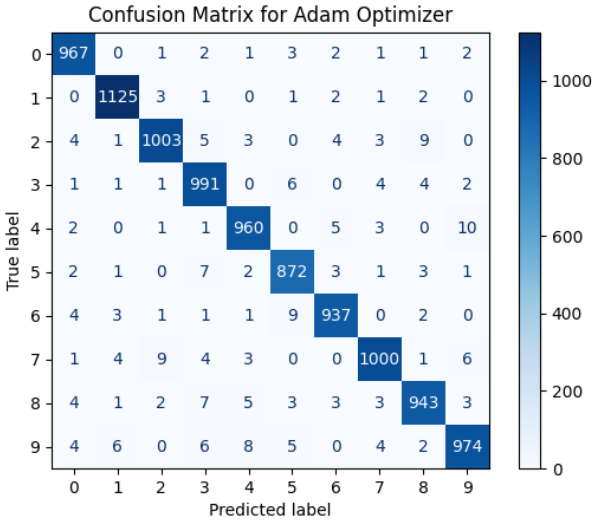
Podľa grafov vieme posúdiť, že SGD s momentom 0,9 pracuje efektívnejšie ako SGD. Začalo trénovať s chybou 2,2899 a skočilo pri 0,3069. Testovať začalo s chybou 2,2737 a skončilo pri s chybnosťou 0,2951. Celková presnosť trénovania bola 91,22% a presnosť testovania 91,36%. Aj keď nebol natrénovaní až na 97%, ale z confusion matrix pekne vidieť, že sa síce mýlil, ale aj tak sa vo všetkých prípadoch trafil a vyhodnotil obrázok správne.

ADAM – Adaptive Moment Estimation

Adaptívny optimalizačný algoritmus, ktorý kombinuje výhody SGD s momentom a RMSProp. Automaticky upravuje rýchlosť učenia pre každú váhu na základe prvého a druhého momentu gradientov.







Adam bol ako jediný vytrénovaný nad 97%. Začínal s tréningovou chybou 0,9203 a testovacou chybou 0,3839. Na konci bola tréningová chyba rovná iba 0,0217 a testovacia chyba 0,0808. Presnosť trénovania dosiahla až 99,52% a presnosť testovania 97.72%. Z confusion matrix jasne vidieť ako sa Adam minimálne pomýlil a všetky číslice na obrázkoch zanalyzoval správne s najmenšími odchýlkami.

Vyhodnotenie MNIST klasifikácie

Optimazer	Training loss	Testing Loss	Training Accuracy	Testing Accuracy
SGD	1,7231	1,698	63,65%	64,05%
SGD s momentom	0,3069	0,2951	91,22%	91,36%
Adam	0,0217	0,0808	99,52%	97,72%

Z tabuľky môžeme posúdiť, že optimalizačný algoritmus Adam vychádza ako najviac efektívny, pretože má najmenšie straty pri trénovaní a testovaní, a podarilo sa ho vytrénovať až na 99,52% pri 50 epochách.

Tiež keď sa pozrieme na výsledné Confusion Matrix pre každý algoritmus, tak pekne vidíme, že Adam sa najmenej pomýlil a podarilo sa ho vytrénovať až na 99,52%.

Backpropagation Algoritmus

Zadanie

Úlohou je implementovať algoritmus backpropagation, ktorý umožňuje neurónovej sieti sa učiť pomocou minimalizácie chybovej funkcie. Je potrebné implementovať funkciu forward aj backward pre jednotlivé operátory.

Cieľ zadania

Cieľom zadania je naprogramovať daný algoritmus iba s použitím knižnice NumPy. Implementácia má pozostávať z lineárnej vrstvy, troch aktivačných vrstiev (sigmoid, tnh, relu) a chybovej funkcie MSE – mean squared error.

Očakávaný výstup

Výstupom by mali byť grafy zobrazujúce výsledky pre AND, OR a XOR s jednou vrstvou a dvoma vrstvami a graf pre XOR, kde sa bude meniť momentum a learning rate.

Riešenie

Reprezentácia dát

Rezprezentáciu dát som zvolila cez triedy. Vytvorila som si triedu na lineárnu vrstvu a následne pre každú aktivačnú funkciu.

```
# implementácia dát
class Linear:  4 usages
    # inicializácia váh a biasov
    def __init__(self, input_size, output_size):
        self.weights = np.random.uniform(size=(input_size, output_size))
        self.bias = np.random.uniform(size=(1, output_size))

    # metóda forward
    def forward(self, X):  6 usages (2 dynamic)
        self.input = X
        return np.dot(X, self.weights) + self.bias

    # metóda backward
    def backward(self, grad_output, learning_rate):  11 usages (8 dynamic)
        grad_input = np.dot(grad_output, self.weights.T) # gradient na vstupe
        grad_weights = np.dot(self.input.T, grad_output) # gradient na váhach
        grad_bias = np.sum(grad_output, axis=0, keepdims=True) # gradient na biasoch

        self.weights -= learning_rate * grad_weights # update váh
        self.bias -= learning_rate * grad_bias # update bias

        return grad_input
```

`Class Linear` reprezentuje jednu plne prepojenú vrstvu, kedy sa na začiatku inicializujú váhy a bias náhodnými hodnotami. Pomocou funkcie forward sa vypočíta výstup vrstvy ako násobok vstupu a váh a k tomu sa pripočíta bias. Funkcia backward počíta gradienty pre váhy a bias na základe gradientu chyby.

Aktivačné funkcie

Aktivačné funkcie sú reprezentované ako samostatné triedy, kde sa v každej počíta jej derivácia. Hodnoty Sigmoidu sú mapované na interval (0,1), hodnoty Tanh na (-1,1) a funkcia ReLU nie je lineárna a jej záporné hodnoty sú nahradené nulou.

Chybová funkcia

Chybová funkcia MSE (Mean Squared Error) vypočítavá kvadratickú chybu tzv. loss. Následne vo funkcii `mse_derivated` sa táto funkcia MSE zderivuje, aby sa zabránilo spätnému šíreniu chyby. Derivácia MSE zabezpečuje, že gradient chyby (rozdiel $y_{pred} - y_{true}$) sa šíri správnym smerom späť cez sieť.

```python
# altivacne funkcie
class Sigmoid:  2 usages
    @staticmethod  3 usages (2 dynamic)
    def activate(X):
        X = np.clip(X, -500,  a_max: 500)
        return 1 / (1 + np.exp(-X))

    @staticmethod  3 usages (2 dynamic)
    def derivative(output):
        output = np.clip(output,  a_min: 1e-7, 1 - 1e-7)
        return output * (1 - output)

class Tanh:  1 usage
    @staticmethod  2 usages (2 dynamic)
    def activate(X):
        X = np.clip(X, -500,  a_max: 500)
        return np.tanh(X)

    @staticmethod  2 usages (2 dynamic)
    def derivative(output):
        output = np.clip(output, -1 + 1e-7, 1 - 1e-7)
        return 1 - output ** 2

class ReLU:  1 usage
    @staticmethod  2 usages (2 dynamic)
    def activate(X):
        return np.maximum(0, X)

    @staticmethod  2 usages (2 dynamic)
    def derivative(output):
        return (output > 0).astype(float)

# Loss function
def mse(y_true, y_pred):  1 usage
    return np.mean((y_true - y_pred) ** 2)

# Loss function derivative
def mse_derivative(y_true, y_pred):  1 usage
    return 2 * (y_pred - y_true) / y_true.size
```

Definovanie siete

```python
class NeuralNetwork:  1 usage
    def __init__(self, input_size, hidden_size1, output_size, hidden_size2=None, activation='tanh', momentum=momentum):
        np.random.seed(42)
        self.hidden_size2 = hidden_size2
        self.momentum = momentum

        # vyber aktivacnej funkcie
        activation_map = {'sigmoid': Sigmoid, 'tanh': Tanh, 'relu': ReLU}
        self.activation = activation_map[activation]

        # inicializácia vrstiev
        self.layer1 = Linear(input_size, hidden_size1)
        if hidden_size2:
            self.layer2 = Linear(hidden_size1, hidden_size2)
            self.output_layer = Linear(hidden_size2, output_size)
        else:
            self.output_layer = Linear(hidden_size1, output_size)

        # inicializácia aktivacnej funkcie pre vystup
        self.output_activation = Sigmoid()

        # incializácia momenta
        self.velocity_w1 = np.zeros_like(self.layer1.weights)
        self.velocity_b1 = np.zeros_like(self.layer1.bias)
        if hidden_size2:
            self.velocity_w2 = np.zeros_like(self.layer2.weights)
            self.velocity_b2 = np.zeros_like(self.layer2.bias)
        self.velocity_wo = np.zeros_like(self.output_layer.weights) # wo = weights_output
        self.velocity_bo = np.zeros_like(self.output_layer.bias) # bo = bias_output
```

Najprv sa vyberie aktivačná vrstva, ktorá bude použitá. Následne sa inicializujú vrstvy siete, a to tak, že prvá vrstva prepojí vstupy s prvou skrytou vrstvou a **výstupná vrstva mapuje výstupy poslednej skrytej vrstvy na konečné výstupy**. Ak existuje aj druhá skrytá vrstva, tak sa na ňu prepoja výstupy z prvej skrytej vrstvy. Pridaním momenta sa urýchli konvergencia. Následne sa váhy a biasy nastavia na nulové hodnoty pre začiatok výpočtu momenta.

Forward a backward metóda

```python
class NeuralNetwork:  1 usage
    # metóda forward
    def forward(self, X):  3 usages (2 dynamic)
        self.hidden1 = self.layer1.forward(X)
        self.activated_hidden1 = self.activation.activate(self.hidden1)
        if self.hidden_size2:
            self.hidden2 = self.layer2.forward(self.activated_hidden1)
            self.activated_hidden2 = self.activation.activate(self.hidden2)
            self.output = self.output_layer.forward(self.activated_hidden2)
        else:
            self.output = self.output_layer.forward(self.activated_hidden1)
        self.predicted_output = self.output_activation.activate(self.output)
        return self.predicted_output

    # metóda backward
    def backward(self, X, y, learning_rate):  9 usages (8 dynamic)
        # Compute the error at the output
        error = mse_derivative(y, self.predicted_output)
        grad_output = error * self.output_activation.derivative(self.predicted_output)  # gradient na vystupe

        if self.hidden_size2:
            # Backpropagation na druhej vrstve
            grad_hidden2 = self.output_layer.backward(grad_output, learning_rate)
            grad_hidden2_activated = self.activation.derivative(self.hidden2) * grad_hidden2
            grad_hidden1 = self.layer2.backward(grad_hidden2_activated, learning_rate)
        else:
            grad_hidden1 = self.output_layer.backward(grad_output, learning_rate)

        grad_hidden1_activated = self.activation.derivative(self.hidden1) * grad_hidden1

        # update váh a biasov pomocou momenta
        self.velocity_wo = self.momentum * self.velocity_wo + learning_rate * np.dot(self.activated_hidden1.T, grad_output)
        self.velocity_bo = self.momentum * self.velocity_bo + learning_rate * np.sum(grad_output, axis=0, keepdims=True)
        self.output_layer.weights -= self.velocity_wo
        self.output_layer.bias -= self.velocity_bo

        if self.hidden_size2:
            self.velocity_w2 = self.momentum * self.velocity_w2 + learning_rate * np.dot(self.activated_hidden1.T, grad_hidden2_activated)
            self.velocity_b2 = self.momentum * self.velocity_b2 + learning_rate * np.sum(grad_hidden2_activated, axis=0, keepdims=True)
            self.layer2.weights -= self.velocity_w2
            self.layer2.bias -= self.velocity_b2

        self.velocity_w1 = self.momentum * self.velocity_w1 + learning_rate * np.dot(X.T, grad_hidden1_activated)
        self.velocity_b1 = self.momentum * self.velocity_b1 + learning_rate * np.sum(grad_hidden1_activated, axis=0, keepdims=True)
        self.layer1.weights -= self.velocity_w1
        self.layer1.bias -= self.velocity_b1
```

Metóda forward

Metóda forward slúži na výpočet výstupov siete pre zadané vstupy. Prenáša vstupy cez prvú vrstvi a aplikuje aktivačnú funkciu. Ak existuje druhá skrytá vrstva, tak sa prenesie už aktivovaný výstup prvej vrstvy cez druhú vrstvu a znova sa použije aktivačná funkcia. Výstup prechádza výstupnou funkciou sigmoid, aby sa hodnoty mapovali do intervalu (0,1).

Metóda backward

Backward metóda aktualizuje váhy a biasy na základe chyby. Vypočíta chybu na výstupe siete pomocou derivácie MSE a následne začína od výstupnej vrstvy a postupne prepočítava gradienty cez vrstvy – backpropagation. Na aktualizácie váh a biasov používa gradienty a momentum, aby zmenila ich hodnoty vo všetkých vrstvách siete.

Funkcia na trénovanie

```python
# funkcia na trénovanie modelu
def train_model(X, y, hidden_size1, epochs, learning_rate, hidden_size2=None, activation='tanh', momentum=momentum):
    input_size = X.shape[1]
    output_size = y.shape[1]
    model = NeuralNetwork(input_size, hidden_size1, output_size, hidden_size2, activation, momentum)
    train_losses = []
    for epoch in range(epochs):
        model.forward(X)
        mse_error = mse(y, model.predicted_output)
        train_losses.append(mse_error)
        model.backward(X, y, learning_rate)
        if (epoch + 1) % 100 == 0:
            print(f'Epoch {epoch + 1}, MSE: {mse_error:.4f}')
    return train_losses, model
```

Táto funkcia slúži na vytrénovania neurónovej siete pomocou zadaných vstupov X a cieľových výstupov y. Trénovanie pozostáva z opakovaných forward a backward výpočtov. Na začiatku sa zistia rozmery podľa vstupov a výstupov a vytvorí sa tak neurónová sieť pomocou triedy NeuralNetwork. Trénovací cyklus pozostáva z epóch (v tomto prípade bola trénovaná na 500 epochách). Každá epocha prejde forward metódou, potom sa vypočíta chyba medzi predpovedaným výstupom a cieľovým výstupom pomocou funkcie MSE a následne sa aktualizujú váhy a biasy pomocou metódy bacward.

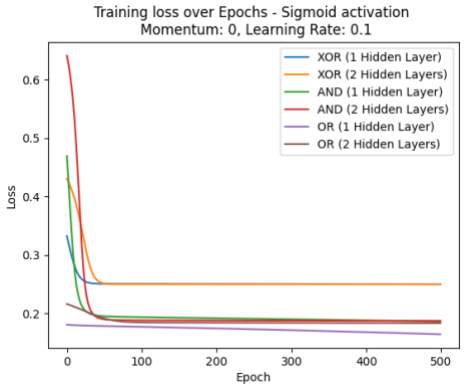
Na vstup a výstup boli použité nasledujúce parametre:

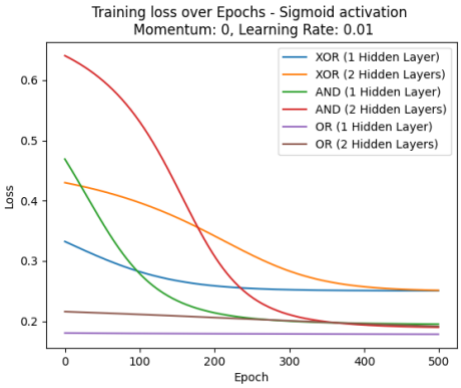
```python
# vstup and vystup
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_XOR = np.array([[0], [1], [1], [0]])
y_AND = np.array([[0], [0], [0], [1]])
y_OR = np.array([[0], [1], [1], [1]])
```

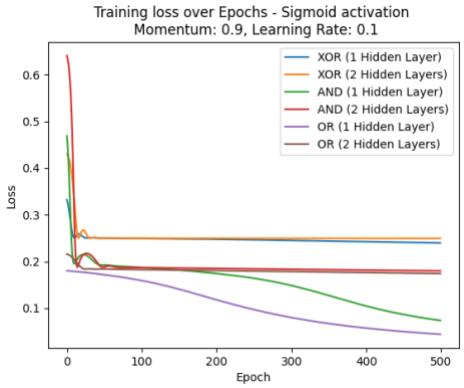
Vyhodnotenie algoritmu Backpropagation

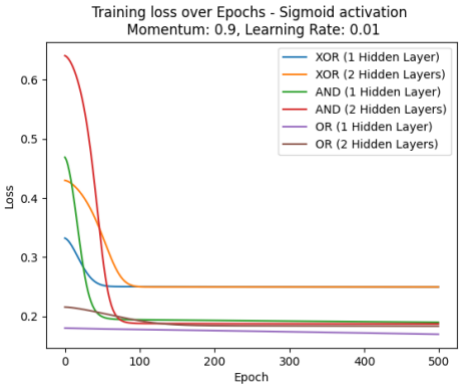
Pri všetkých grafoch som skúšala meniť momentum (0 alebo 0,9) a learning rate (0,01 a 0,1), aby som porozovala, ako rýchlo a efektívne sa neurónová sieť dokáže naučiť.

Sigmoid

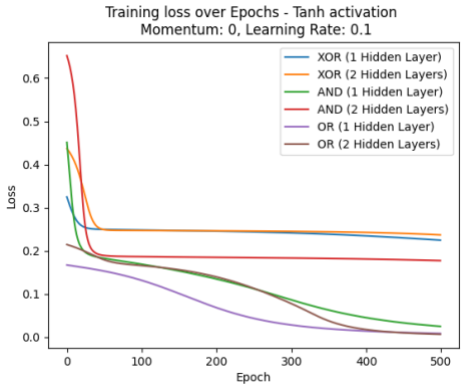


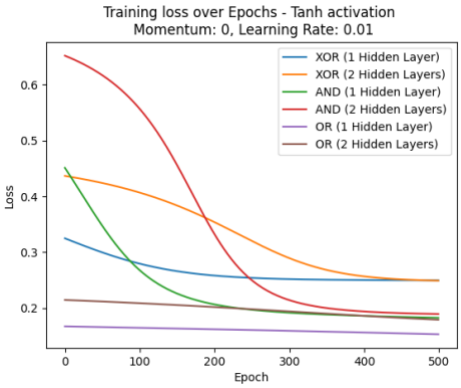


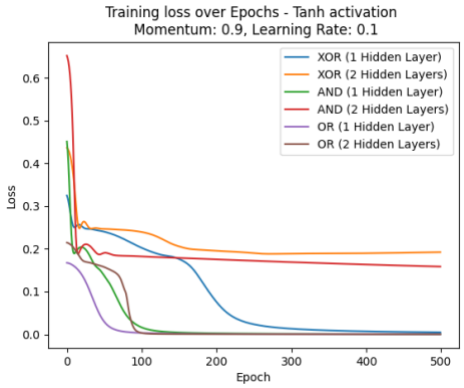


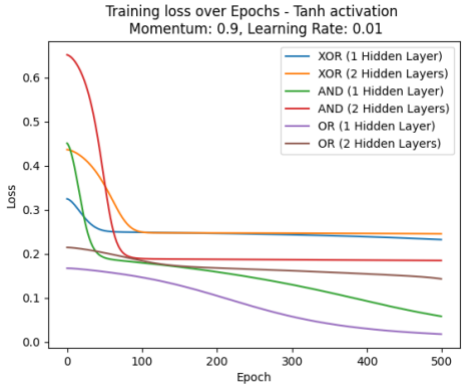


Tanh

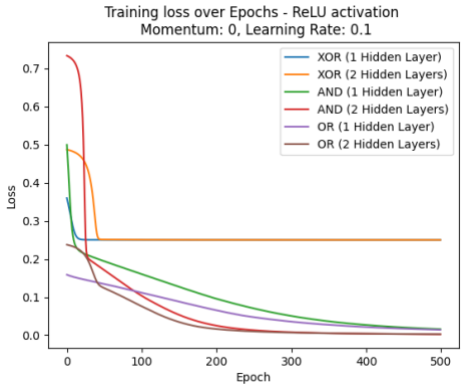


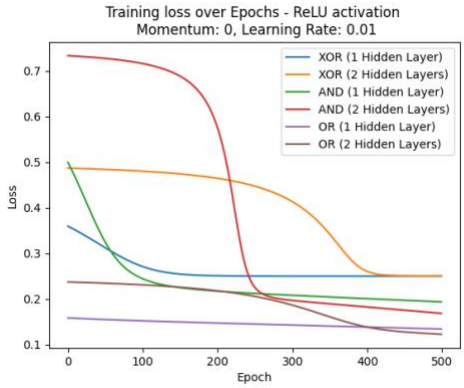


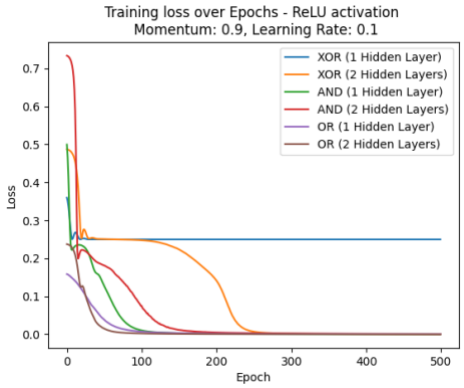


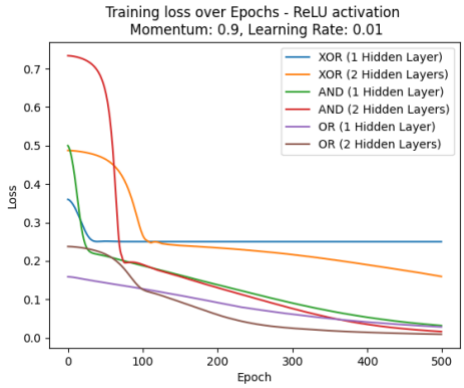


ReLU









Už len po zhliadnutí samotných grafov je jasne viditeľné:

- s momentom sa učenie zrýchľuje a dosahuje stabilnejšie výsledky
- vyššia rýchlosť učenia (0,1) vedie k rýchlejšej konvergencii
- nižšia rýchlosť učenia (0,01) je stabilnejšia
- pridaním druhej vrstvy sa zlepšuje výkon

Pri funkcii XOR môžeme sledovať, že je výsledok je viac optimálny, keď prechádza dvoma skrytými vrstvami a najviac efektívna pre jeho trénovanie je aktivačná funkcia ReLU s momentom 0,9 a learning rate 0,1 lebo vtedy dokázal dosiahnúť aj stratovú hodnotu 0.