Zadanie 3A Dokumentácia

MNIST klasifikátor

Popis architektúry

```
transform = Compose([
    ToTensor(),  # Konvertuje obrázky na tenzory a škáluje hodnoty do intervalu [0, 1]
])

# Načítanie MNIST datasetu (číslice 0-9)
train_data = datasets.MNIST(root='data', train=True, transform=transform, download=True)
test_data = datasets.MNIST(root='data', train=False, transform=transform, download=True)

# DataLoaders na dávkovanie tréningových a testovacích dát
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)  # Shuffle pre generalizáciu
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)  # Bez shuffle pre konzistenciu
```

Najskôr začneme načítaním datasetu a použijeme ToTensor() aby sme dostali hodnoty v intervale [0,1] a potom si ich rozdelíme do train_data a test_data. Následne inicializujeme DataLoadre, kde pri trénovaní dáta pomiešame(shuffle=True) aby sme sa vyhli tomu že sa neurónová sieť naučí na poradie.

```
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.dropout = nn.Dropout(0.2)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28 * 28, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, 10)

    def forward(self, x):
        x = self.flatten(x)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = torch.relu(self.fc2(x))
        x = torch.relu(self.fc3(x))
        x = self.fc4(x)
        return x
```

Ďalej nasleduje definícia viacvrstvového perceptónu, kde inicializujeme vrstvy siete a definujeme priechod dopredu. V inicializácii nn:dropout(0.2) pridáva dropout vrstvu s pravdepodobnosťou 20%. Dropout slúži na zníženie nadmerného prispôsobovania (overfitting) tým, že náhodne "vypína" niektoré neuróny počas trénovania. Následne nn. Flatten() prevedie vstup (obrázok 28×28) na jednorozmerné pole s dĺžkou 784. Ďalej nasledujú vrstvy: výstupná a 4 hidden. V metóde forward() x = self.flatten(x) rozvine vstupný obrázok na jednorozmerné pole a potom aplikuje ReLU na lineárne vrstvy a použije dropout na výstup z prvej vrstvy. Nakoniec prevedie dáta cez poslednú lineárnu vrstvu. Táto vrstva generuje skóre pre každú z 10 tried a metóda vráti výstupy zo siete (batch_size,10), kde každý riadok obsahuje skóre pre triedy.

```python
def train(model, loader, optimizer, criterion):
    model.train()  # Nastavenie modelu do tréningového módu
    total_loss = 0
    correct = 0
    for data, target in loader:  # Prechod cez dávky dát
        data, target = data.to(device), target.to(device)  # Prenos na zariadenie (GPU/CPU)
        optimizer.zero_grad()  # Vynulovanie gradientov
        output = model(data)  # Forward pass
        loss = criterion(output, target)  # Výpočet chyby
        loss.backward()  # Spätná propagácia
        optimizer.step()  # Aktualizácia váh
        total_loss += loss.item()  # Akumulácia straty
        pred = output.argmax(dim=1)  # Predikcia: trieda s najvyššou pravdepodobnosťou
        correct += pred.eq(target).sum().item()  # Počet správnych predikcií
    accuracy = 100. * correct / len(loader.dataset)  # Výpočet presnosti
    return total_loss / len(loader), accuracy  # Priemerná strata a presnosť
```

model.train()
Nastavuje model do tréningového módu.
total_loss = 0 a correct = 0
Inicializuje premenné na sledovanie celkovej straty a počtu správnych predikcií počas tréningového cyklu. total_loss sa použije na akumuláciu hodnoty straty, ktorá sa neskôr použije na výpočet priemernej straty.
optimizer.zero_grad()
Vynuluje gradienty pred začatím nového výpočtu. V PyTorch sa gradienty akumulujú, takže je potrebné ich vynulovať pred každým spätným propagovaním.
output = model(data)
Prevedie dáta cez model (vykoná forward pass). Model vráti výstupy, ktoré sú predikcie pre jednotlivé triedy.
loss = criterion(output, target)
Vypočíta stratu medzi predikciami modelu a skutočnými hodnotami..
loss.backward()
Vykoná spätnú propagáciu, ktorá počíta gradienty pre všetky parametre modelu na základe straty.
optimizer.step()
Aktualizuje váhy modelu na základe vypočítaných gradientov. Tento krok sa vykonáva po výpočte gradientov a spätnom šírení.
total_loss += loss.item()
Akumuluje hodnotu straty pre túto dávku dát. loss.item() získava hodnotu straty ako číslo (bez gradientov).
pred = output.argmax(dim=1)
Predikcia triedy pre každý vstup je určená ako trieda s najvyššou hodnotou výstupu modelu. argmax(dim=1) vracia index triedy s najvyššou pravdepodobnosťou.
correct += pred.eq(target).sum().item()
Počíta počet správnych predikcií. pred.eq(target) vytvorí tensor, kde je True pre správne predikcie a False pre nesprávne. sum() spočíta počet správnych predikcií.

Po ukončení cyklu sa vypočíta presnosť, ktorá je pomerom správnych predikcií k celkovému počtu príkladov v trénovacej sade.
Funkcia vracia priemernú stratu počas celej epochy. Presnosť modelu, ktorá je percentuálnym podielom správnych predikcií z celkového počtu príkladov.

```python
def test(model, loader, criterion):
    model.eval()  # Nastavenie modelu do evaluačného módu
    total_loss = 0
    correct = 0
    with torch.no_grad():  # Deaktivácia výpočtu gradientov
        for data, target in loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            loss = criterion(output, target)
            total_loss += loss.item()
            pred = output.argmax(dim=1)
            correct += pred.eq(target).sum().item()
    accuracy = 100. * correct / len(loader.dataset)
    return total_loss / len(loader), accuracy
```

Testovacia funkcia sa v podstate správa ako trénovacia až na zopár rozdieľov: Testovacia funkcia používa model.eval(), ktorý nastaví model do evaluačného režimu. V tomto režime sa mechanizmy ako dropout deaktivujú, takže model pracuje bez náhodného vypínania neurónov. Taktiež v testovacej funkcii sa gradienty nepočítajú, pretože počas testovania sa neaktualizujú váhy. A nakoniec V testovacej funkcii sa žiadna aktualizácia váh neuskutočňuje. Je zameraná len na hodnotenie modelu, preto nie je potrebná spätná propagácia ani optimalizácia.

```python
optimizers = {
    "SGD": lambda model: optim.SGD(model.parameters(), lr=0.01),  # Stochastický gradientný zostup
    "SGD_momentum": lambda model: optim.SGD(model.parameters(), lr=0.01, momentum=0.9),  # SGD s momentom
    "Adam": lambda model: optim.Adam(model.parameters(), lr=0.001)  # Adam optimalizátor
}
```

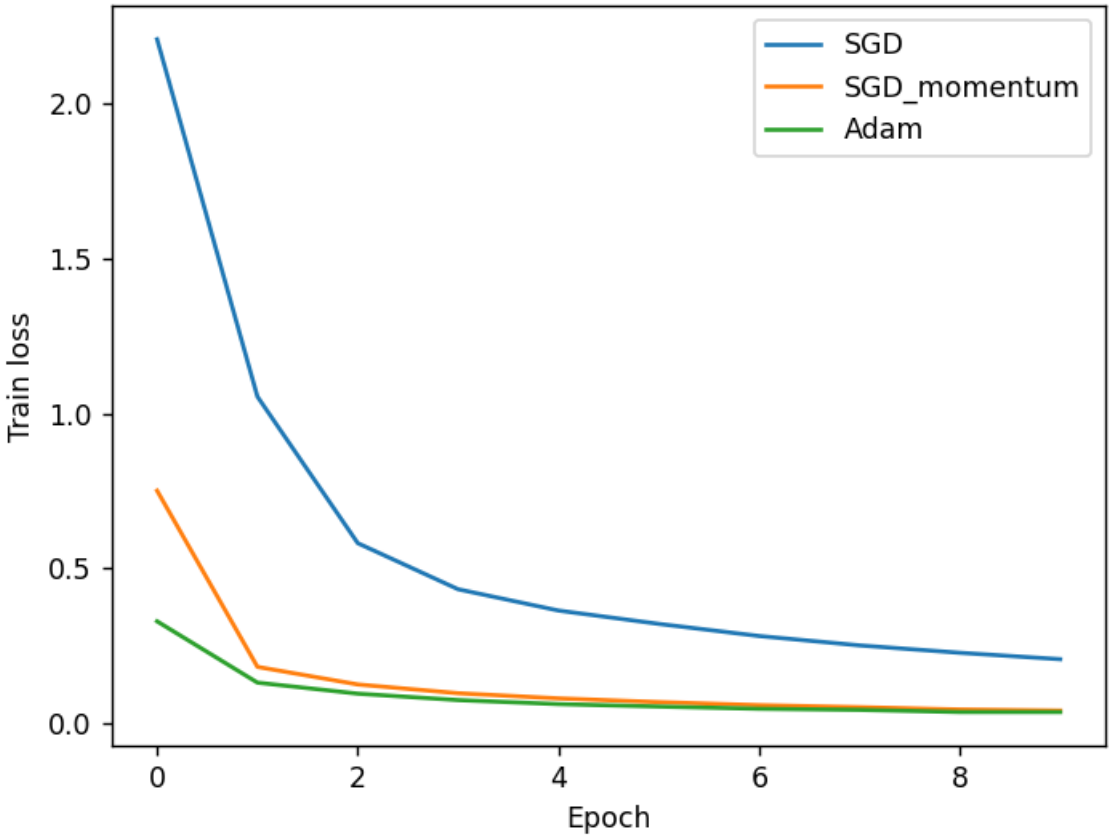
Ďalej si zadefinujeme slovník optimalizátorov v PyTorch, kde každý kľúč predstavuje typ optimalizátora a hodnota je lambda funkcia, ktorá inicializuje príslušný optimalizátor pre daný model.

```python
for opt_name, opt_fn in optimizers.items():
    print(f"Training with {opt_name}...")
    model = MLP().to(device)  # Inicializácia modelu a prenos na zariadenie
    optimizer = opt_fn(model)  # Výber optimalizátora
    criterion = nn.CrossEntropyLoss()  # Funkcia na výpočet chyby

    # Na uloženie priebežných výsledkov
    train_losses, train_accuracies = [], []
    test_losses, test_accuracies = [], []

    # Tréning pre počet epoch
    num_epoch = 10
    for epoch in range(num_epoch):
        train_loss, train_acc = train(model, train_loader, optimizer, criterion)
        test_loss, test_acc = test(model, test_loader, criterion)

        # Ukladanie výsledkov
        train_losses.append(train_loss)
        train_accuracies.append(train_acc)
        test_losses.append(test_loss)
        test_accuracies.append(test_acc)

        # Výpis výsledkov pre aktuálnu epochu
        print(f"Epoch {epoch + 1}: Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.2f}%, "
              f"Test Loss: {test_loss:.4f}, Test Acc: {test_acc:.2f}%")

    # Uloženie výsledkov pre aktuálny optimalizátor
    results[opt_name] = {
        "train_losses": train_losses,
        "train_accuracies": train_accuracies,
        "test_losses": test_losses,
        "test_accuracies": test_accuracies
    }
```

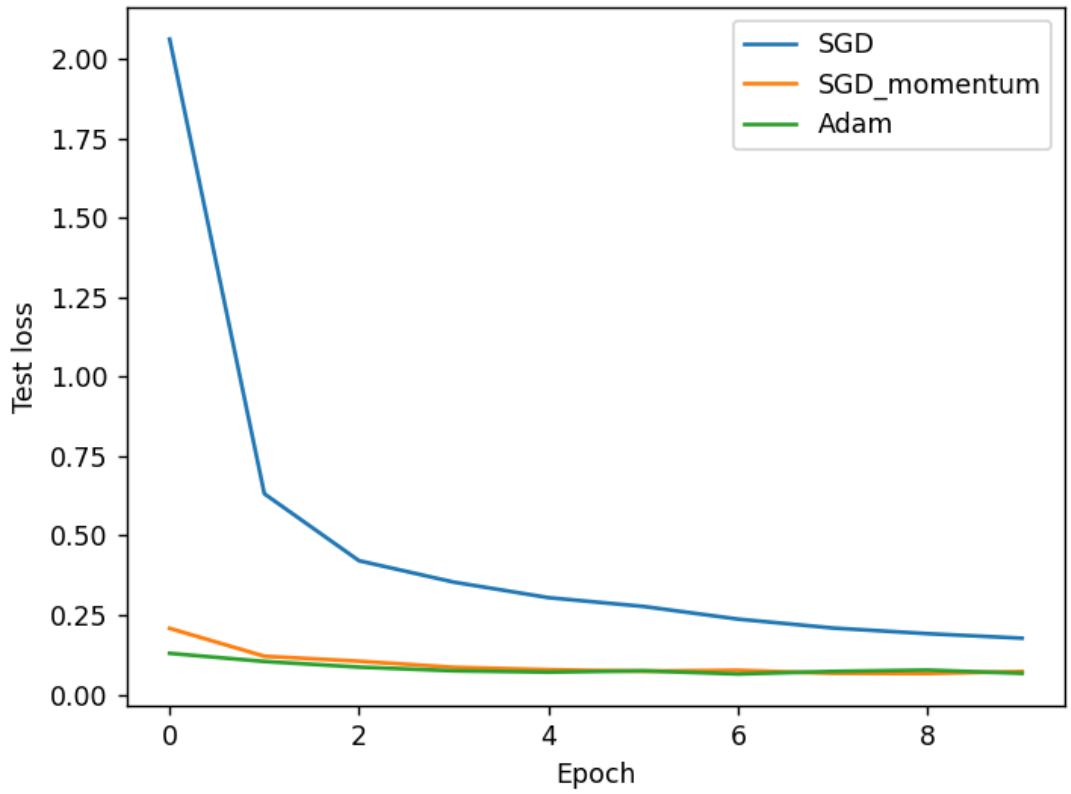
Nakoniec vykonáme tréning a testovanie modelu pre každý optimalizátor definovaný v predchádzajúcom slovníku optimizers. Pre každý optimalizátor sa model trénuje a testuje počas niekoľkých epoch, pričom sa sledujú výsledky (strata a presnosť) počas tréningového aj testovacieho procesu. Na konci sú výsledky uložené pre každý optimalizátor.

Tabuľka hyperparametrov

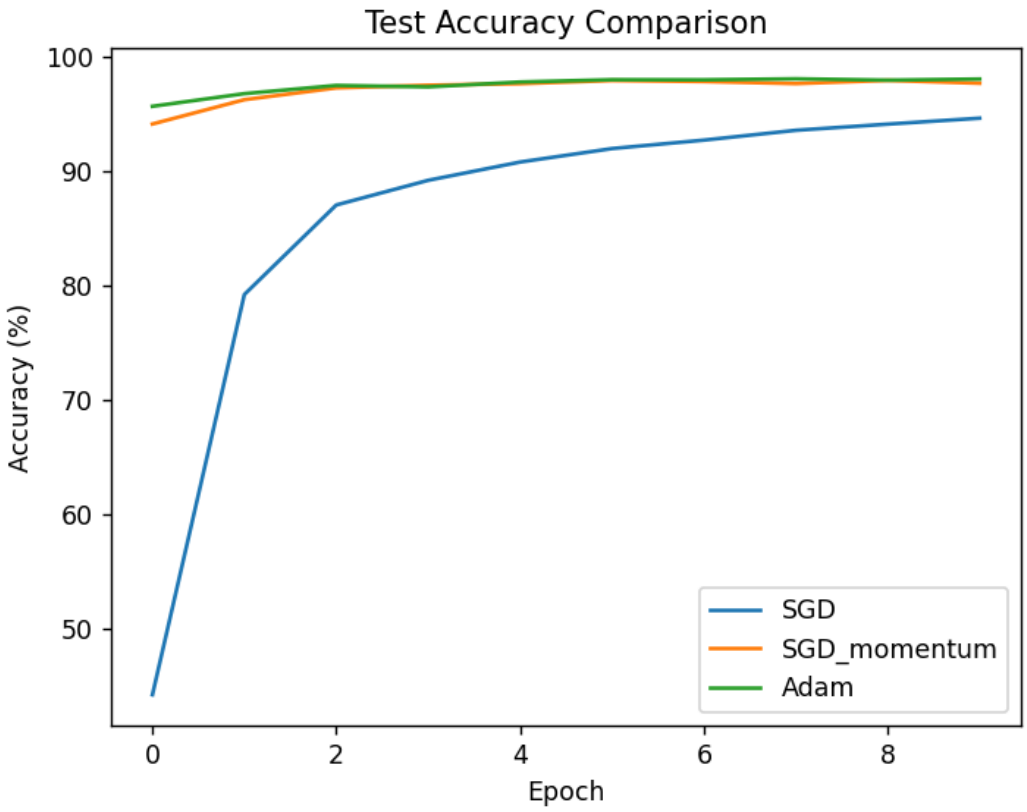
Počet vrstiev	4
Počet neurónov na vrstve	256, 128, 64
Optimizers	SGD, SGD_momentum
Learning rate	SGD: 0.01, SGD_momentum: 0.01, Adam: 0.001
Momentum	0.9 pre SGD_momentum
Batch size	64
Počet epoch	10
Aktivačné funkcie	ReLU
Dropout rate	0.2
Počiatočné váhy	Automatická inicializácia cez nn.Linear



Tu môžeme vidieť vývoj trénovacej chyby pre všetky tri optimalizátory.



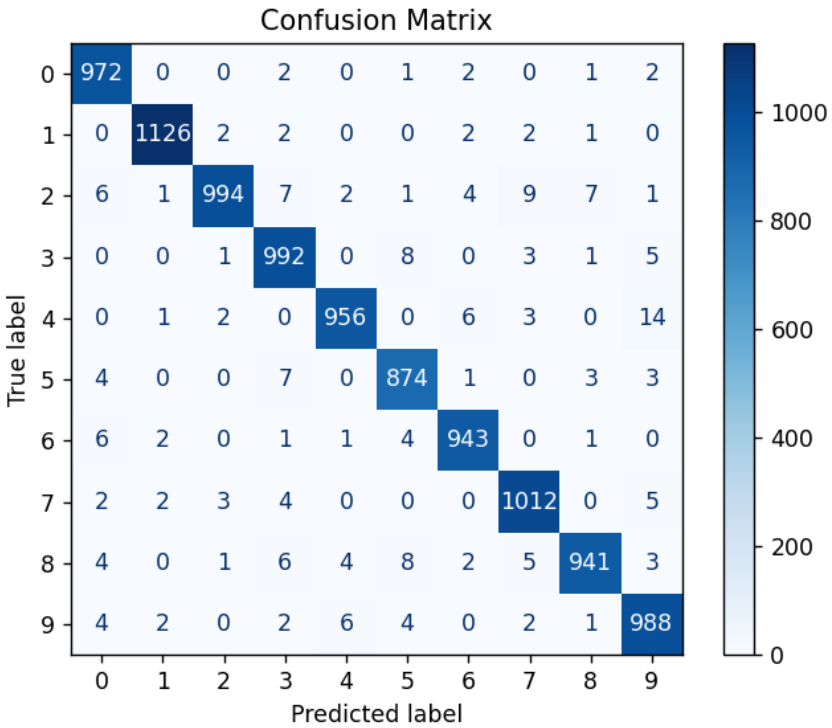
Tu môžeme vidieť vývoj testovacej chyby pre všetky tri optimalizátory.



Tu môžeme vidieť vývoj presnosti pre všetky tri optimalizátory.

Vyhodnotenie

Confusion Matrix pre najlepší algoritmus Adam:



Zhodnotenie

Vyhodnotil by som Adam optimalizačný algoritmus za najlepší. Pri mojom testovaní ukazoval vždy najväčšie percento presnosti či už pri trénovaní alebo testovaní, mal najmenší test loss a najmenší train loss. Ako druhý najlepší by som vyhodnotil SGD_momentum, ktoré nebolo o moc horšie ako Adam. Ale SGD bolo jasným najhorším oprimalizátorom.

Backpropagation algoritmus

Implementácia

```python
class Linear:
    def __init__(self, input_dim, output_dim):
        self.weights = np.random.uniform(-1, 1, (input_dim, output_dim))
        self.biases = np.zeros((1, output_dim))
        self.input = None
        self.grad_weights = None
        self.grad_biases = None

    def forward(self, x):
        self.input = x
        return np.dot(x, self.weights) + self.biases

    def backward(self, grad_output):
        self.grad_weights = np.dot(self.input.T, grad_output)
        self.grad_biases = np.sum(grad_output, axis=0, keepdims=True)
        return np.dot(grad_output, self.weights.T)

    def update_params(self, lr, momentum, grad_weights_prev, grad_biases_prev):
        update_weights = momentum * grad_weights_prev - lr * self.grad_weights
        update_biases = momentum * grad_biases_prev - lr * self.grad_biases

        self.weights += update_weights
        self.biases += update_biases

        return update_weights, update_biases
```

Tu definujeme triedu Linear, ktorá implementuje lineárnu vrstvu. V prvom kroku, v metóde __init__, sú inicializované váhy a biasy. Váhy sa generujú náhodne v rozsahu od -1 do 1 s rozmermi, ktoré zodpovedajú počtu vstupov (input_dim) a počtu výstupov (output_dim). Biasy sú inicializované na nulu. Metóda forward je zodpovedná za výpočet výstupu vrstvy. Na vstupe dostane hodnoty x (vstupné dáta), ktoré sú maticou s tvarom (batch_size, input_dim). Tento vstup je transformovaný pomocou lineárnej funkcie: výstup sa vypočíta ako súčet váh násobených vstupmi a biasov. Výstup tejto transformácie je maticou tvaru (batch_size, output_dim). V metóde backward sa počítajú gradienty váh a biasov, ktoré budú použité na ich aktualizáciu počas učenia. Metóda dostáva ako vstup gradient straty vzhľadom na výstup (označovaný ako grad_output). Na základe tohto gradientu sa vypočíta gradient váh a biasov. Gradienty váh sa počítajú ako skalárny súčin transponovaného vstupu a gradientu výstupu, zatiaľ čo gradienty biasov sa získavajú ako súčet gradientu výstupu. Tiež sa vypočíta gradient vzhľadom na vstup, ktorý sa vráti, aby mohol byť použitý pri spätnom šírení v predchádzajúcich vrstvách.

Nakoniec, metóda update_params slúži na aktualizáciu parametrov siete (váh a biasov). Aktualizácie sa robia pomocou gradientného zostupu, pričom sa používajú aj momentum. Na základe zadaných hodnôt pre rýchlosť učenia (lr) a momentum, sa váhy a biasy upravia tak, aby minimalizovali chybu modelu.

Aktivačné funkcie:

```
class Sigmoid:
    def __init__(self):
        self.output = None

    def forward(self, x):
        self.output = 1 / (1 + np.exp(-x))
        return self.output

    def backward(self, grad_output):
        return grad_output * self.output * (1 - self.output)

# ReLU funkcia
class ReLU:
    def __init__(self, alpha=0.01):
        self.alpha = alpha
        self.output = None

    def forward(self, x):
        self.output = np.where(x > 0, x, self.alpha * x)
        return self.output

    def backward(self, grad_output):
        grad = np.where(self.output > 0, 1, self.alpha)
        return grad_output * grad

# Tanh funkcia
class Tanh:
    def __init__(self):
        self.output = None

    def forward(self, x):
        self.output = np.tanh(x)
        return self.output

    def backward(self, grad_output):
        return grad_output * (1 - self.output ** 2)
```

MSE:

```python
class MSE:
    def __init__(self):
        self.y_true = None
        self.y_pred = None

    def forward(self, y_true, y_pred):
        self.y_true = y_true
        self.y_pred = y_pred
        return np.mean((y_true - y_pred) ** 2)

    def backward(self):
        return -2 * (self.y_true - self.y_pred) / self.y_true.shape[0]
```

Metóda forward vypočíta hodnotu MSE na základe skutočných a predpovedaných hodnôt a vracia skalár predstavujúci priemernú chybu. **Metóda backward v**ypočíta gradient funkcie MSE vzhľadom na predpovedané hodnoty a vracia maticu alebo vektor gradientov s rovnakým tvarom ako vstupy.

```python
class Model:
    def __init__(self, layers):
        self.layers = layers
        self.momentum_weights = [np.zeros_like(layer.weights) if hasattr(layer, 'weights') else None for layer in layers]
        self.momentum_biases = [np.zeros_like(layer.biases) if hasattr(layer, 'biases') else None for layer in layers]

    def forward(self, x):
        for layer in self.layers:
            x = layer.forward(x)
        return x

    def backward(self, loss_grad):
        for layer in reversed(self.layers):
            loss_grad = layer.backward(loss_grad)

    def update_params(self, lr, momentum):
        for i, layer in enumerate(self.layers):
            if hasattr(layer, 'weights'):
                self.momentum_weights[i], self.momentum_biases[i] = layer.update_params(
                    lr, momentum, self.momentum_weights[i], self.momentum_biases[i]
                )
```

__init__(): Pre každú vrstvu sa vytvoria nulové matice (rovnakej veľkosti ako váhy a biasy vrstvy), ktoré budú uchovávať momenta pre aktualizácie váh a biasov. Tieto momenta sú inicializované len pre vrstvy, ktoré obsahujú váhy a biasy.

Forward(): Každá vrstva aplikovaná na vstup transformuje dáta podľa svojej funkcie (napr. lineárna transformácia alebo aktivačná funkcia). Výstup jednej vrstvy sa stáva vstupom pre nasledujúcu.

Backward(): Gradient chyby sa postupne šíri od poslednej vrstvy po prvú. Každá vrstva vypočíta gradient vzhľadom na svoje vstupy a tieto gradienty sa odovzdajú nasledujúcej vrstve.

Update_params(): Pre vrstvy, ktoré obsahujú váhy a biasy, sa aktualizujú parametre: Použijú sa predchádzajúce hodnoty momenta, aktuálne gradienty a rýchlosť učenia na výpočet zmien váh a biasov a aktualizované momenta sa uložia na ďalšiu iteráciu.

Trénovanie a testovanie:

```python
def train_and_test(X, y, epochs, lr, momentum, model, criterion, problem_name):
    losses = []
    for epoch in range(epochs):
        y_pred = model.forward(X)
        loss = criterion.forward(y, y_pred)
        losses.append(loss)
        loss_grad = criterion.backward()
        model.backward(loss_grad)
        model.update_params(lr, momentum)
        if epoch % 50 == 0 or epoch == epochs - 1:
            print(f"Epoch {epoch}, Loss: {loss:.4f}")

    # Plot the loss graph
    plt.figure(figsize=(8, 5))
    plt.plot(range(epochs), losses, label=f'{problem_name} Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title(f'Training Loss for {problem_name}')
    plt.legend()
    plt.grid(True)
    plt.show()

    y_pred = model.forward(X)
    y_pred_rounded = np.round(y_pred)  # Round predictions
    y_pred_fixed = np.where(y_pred_rounded == -0., 0., y_pred_rounded)  # Fix any -0 values
    print("Predictions:", y_pred_fixed)
```

Vrstvy:

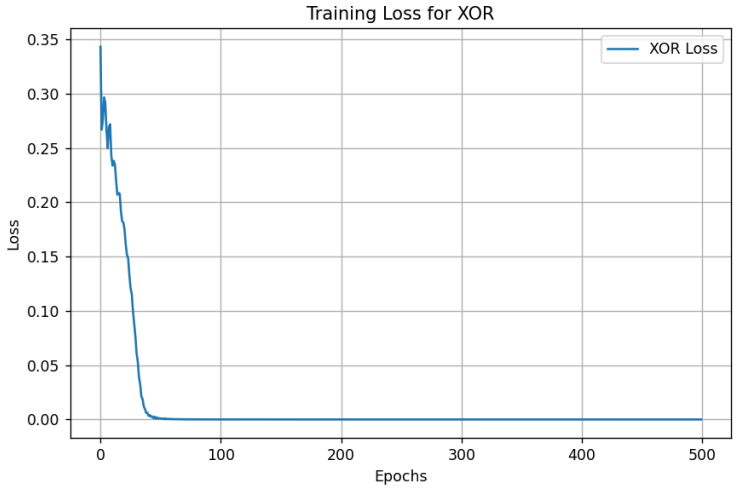
```python
layers = [
    Linear(2, 4),
    Tanh(),
    Linear(4, 4),
    Sigmoid(),
    Linear(4, 1),
    Tanh()
]
```

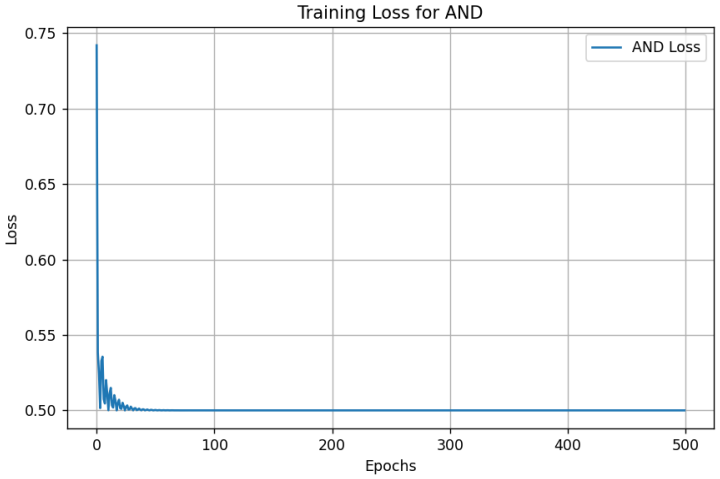
Problémy:

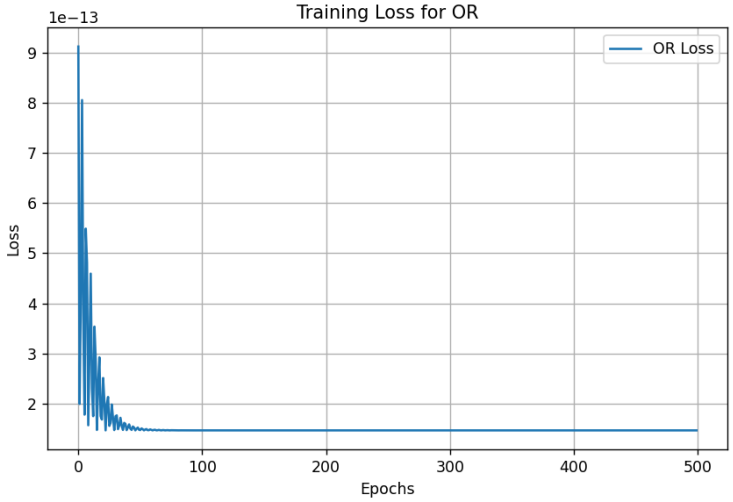
```python
print("XOR Problem:")
X_xor = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_xor = np.array([[0], [1], [1], [0]])
train_and_test(X_xor, y_xor, epochs=500, lr=0.3, momentum=0.9, model=model, criterion=criterion, problem_name="XOR")

print("AND Problem:")
X_and = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_and = np.array([[0], [0], [0], [1]])
train_and_test(X_and, y_and, epochs=500, lr=0.3, momentum=0.9, model=model, criterion=criterion, problem_name="AND")

print("OR Problem:")
X_or = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_or = np.array([[0], [1], [1], [1]])
train_and_test(X_or, y_or, epochs=500, lr=0.3, momentum=0.9, model=model, criterion=criterion, problem_name="OR")
```

Grafy

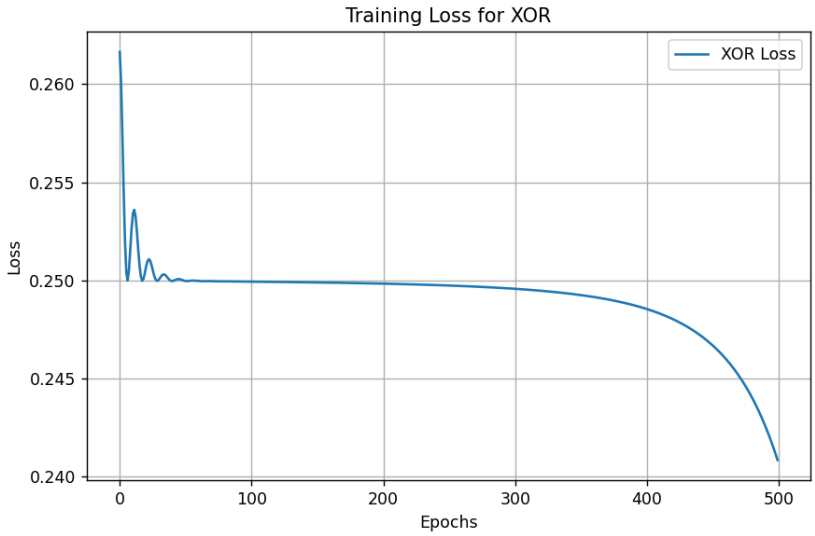
Tanh, 1 vrstva, lr=0.3, momentum=0.9, seed = 124:

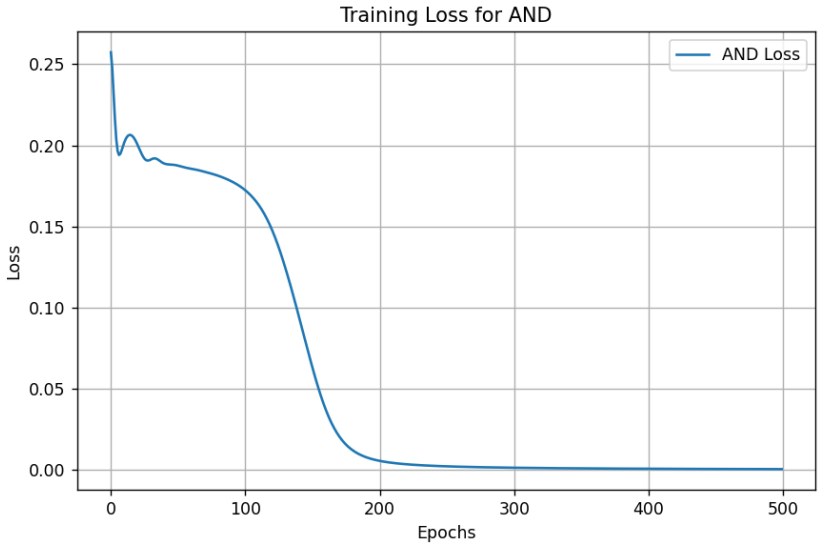


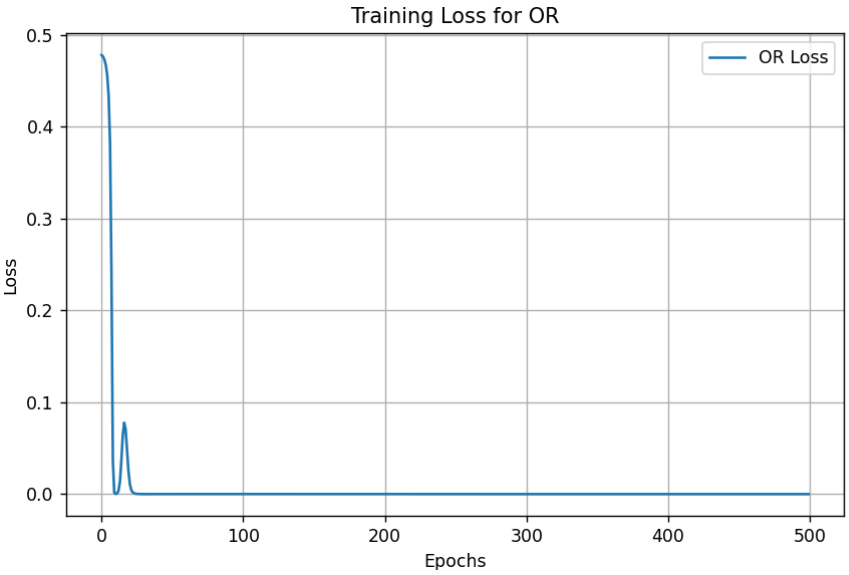




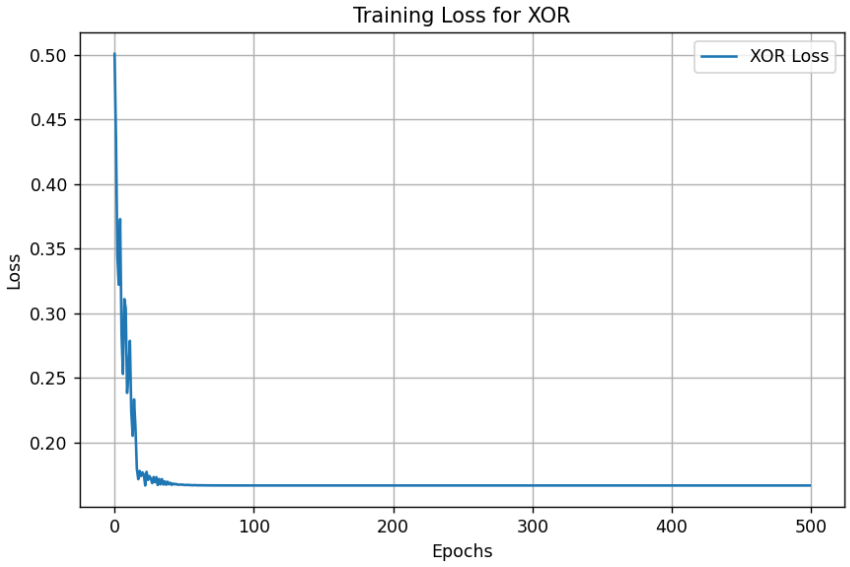
Sigmoid, 2 vrstvy, lr=0.3, momentum=0.9, seed = 123:

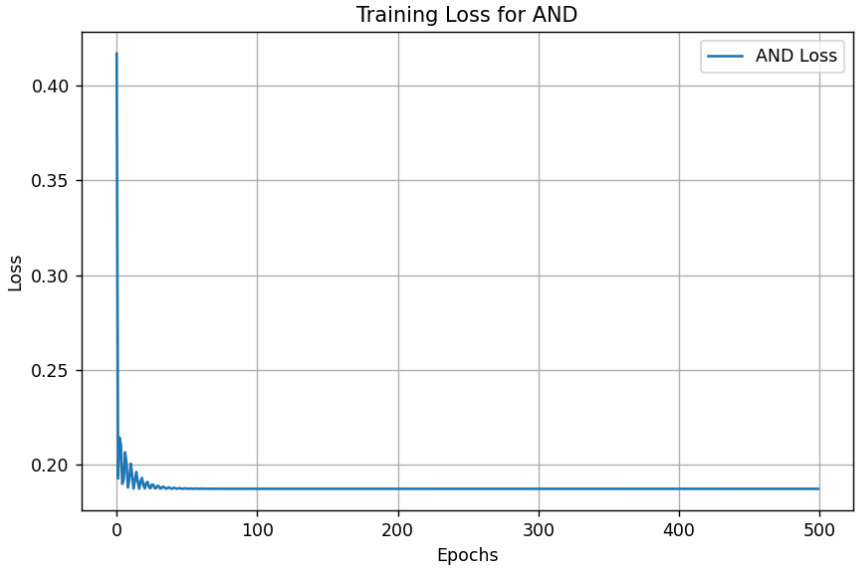


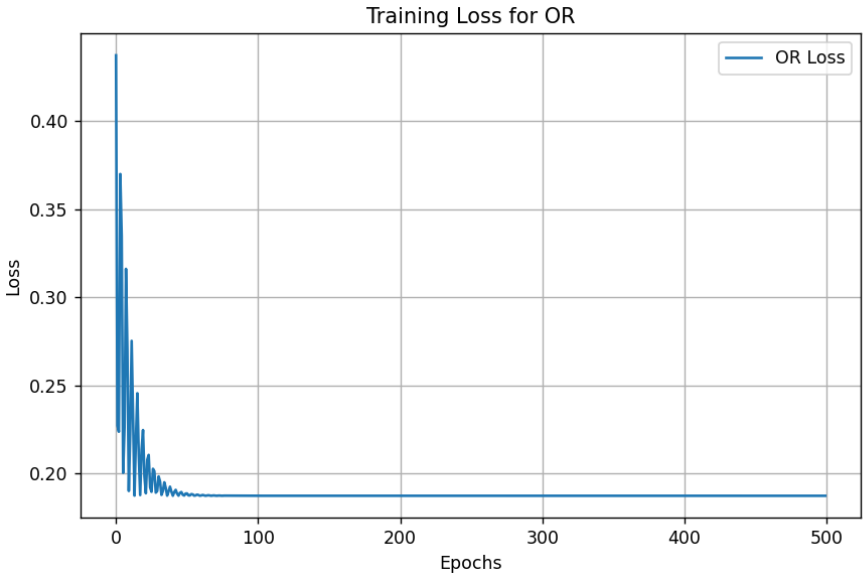




ReLU, 2 vrstvy, lr=0.3, momentum=0.9, seed = 126:

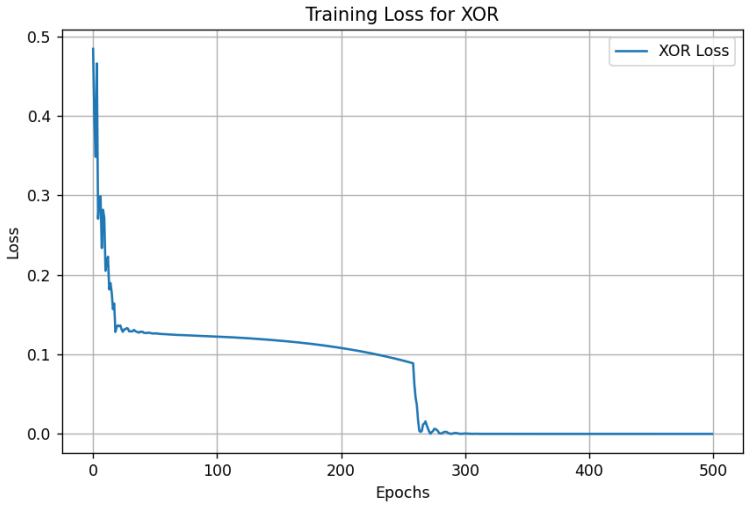




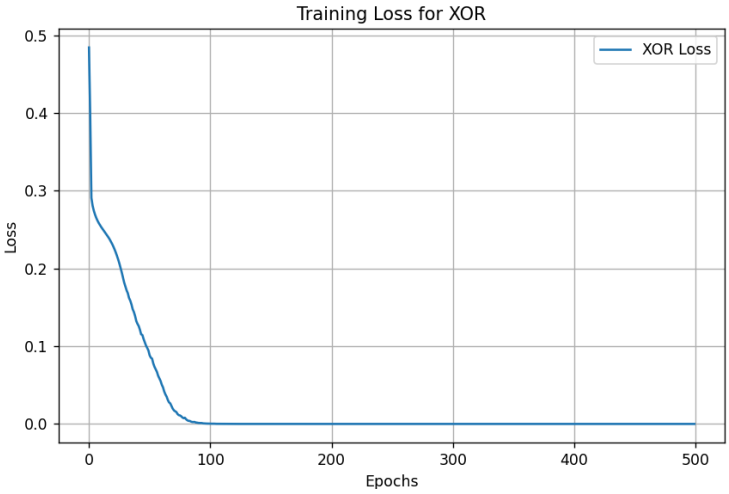


Andrej Krivošík 120955

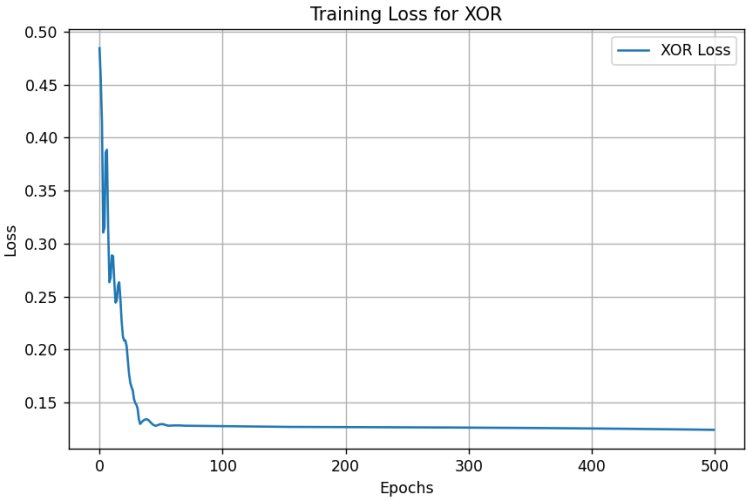
XOR momentum=0.9,lr=0.3



XOR momentum=0, lr =0.3



XOR momentum=0.9,lr=0.1



XOR momentum=0, lr =0.1