RĪGAS TEHNISKĀ UNIVERSITĀTE

Datorzinātnes un informācijas tehnoloģijas fakultāte

Mākslīgā intelekta un sistēmu inženierijas katedra

Rinalds Daniels Pikše

bakalaura akadēmisko studiju programmas "Datorsistēmas"

students, stud. apl. nr. 171RDB359

Salīdzinošā analīze Beiesa neironu tīklu stabilitātei

BAKALAURA DARBS: RUDENS SEMESTRA ATSKAITE

Zinātniskais vadītājs Dr.Sc.Ing. Pētnieks

Ēvalds Urtāns

RĪGA 2022

SATURA RĀDĪTĀJS

1. DARBA BŪTĪBA UN AKTUALITĀTE

Bakalaura darba mērķis: Salīdzināt Beiesa neironu tīklus ar parastajiem neironu tīkliem, izmantojot datu kopu, kura ir ārpus problēmsfēras

Bakalaura darba uzdevumi:

Izprast kā Beiesa neironu tīkli atšķiras no parastiem neironu tīkliem.

Iepazīties ar Monte-Carlo Dropout metodi

Noteikt kā Beiesa neironu tīkli nosaka modeļa nenoteiktību.

Noteikt kā palielināt Beiesa tīklu rezultātu ticamību.

Problēmas nostādne: Neironu tīkli var būt kļūdaini norādot rezultātus datu kopām, kas atrodas ārpus problēmsfēras, kurai neironu tīkls tika trenēts.

Tēmas aktualitātes pamatojums:

Mašīnapmācība ir datorzinību lauks, kas mēģina izstrādāt automatizētus modeļus, lai risinātu plaša spektra problēmas. Mašīnapmācība pēdējos gados ir strauji augusi popularitātē un pielietojumu daudzveidībā dažādās sfērās – tiek aplēsts, ka servisu kopums kas balstīti uz mašīnapmācību varētu vairāk kā desmitkāršoties līdz 2030. gadam(*Machine Learning as a Service Market Size, Report 2030*, 2022). Pielietojot neironu tīklus svarīgi ir zināt, cik liels ir pārliecības līmenis prognozēm. Šis darbs pievēršās Beiesa neironu tīkliem, kuri ir balstīti uz Beiesa metodēm, kuras sāka izplatīties 1980. gados un ir kļuvuši par plaši pielietotu neironu tīklu grupu. Beiesa neironu tīkli ļauj izprast, kādā mērā uzticēties modeļa gala rezultātam - tie izmanto varbūtību sadalījumus mainīgo vietā nevis nominālu vērtības, tādā veidā ievietojot varbūtības aprēķinu katrā modeļa solī. Izpētot tuvāk Beiesa neironu tīklu īpašības, mašīnapmācība var kļūt skaidrāka par veikto prognožu atbilstību problēmu risināšanai. Zināt, kādās robežās uzticēties izveidotajiem modeļiem ir būtiski, jo no tiem tiek veikti lēmumi, kas ietekmē aizvien vairāk cilvēku ikdienā.

1.1. tabula

Bakalaura darba izstrādes plāns

Aktivitāte	Termiņš
Apgūt parasto neironu tīklu teoriju	01.01.2023
Apgūt Beiesa neironu tīklu teoriju, izmantojot Monte-Carlo Dropout metodi	01.02.2023
Apmācīt abu veidu modeļus un veikt eksperimentus	01.03.2023
Dokumentēt rezultātus, formulēt hipotēzes tālākiem pētījumiem	01.04.2023
Bakalaura darba nodošana	29.05.2023

2. INFORMĀCIJAS AVOTU APSTRĀDES REZULTĀTI

2.1. Praktiskie uzdevumi

Pirmā darba aktivitāte bakalaura izstrādei, man ir bijusi "Apgūt parasto neironu tīklu teoriju". Tās ietvaros darba vadītājs, Dr.Sc.Ing. Ēvalds Urtāns, man deva praktiskus uzdevumus, ko pildīt, lai attīstītu savas zināšanas, kas man palīdzēs bakalaura darba izstrādes procesā. Kā pirmos informācijas avotus aprakstīšu katru no šiem uzdevumiem.

2.1.1. Gravitātes spēle

Šī uzdevuma mērķis bija atkārtot python valodas sintaksi, matricu operācijas un iepazīties ar numpy koda bibliotēku. Kopumā mērķi tika sasniegti, sākumā bija neskaidri kā strukturēt kodu un uz ko fokusēties, taču sanāca atkārtot matricu transformācijas un python sintaksi. Pēc koda pārbaudes sapratu, ka svarīgākais algoritma izveidē bija kombinētās matricu transformācijas un numpy bibliotēkai specifiskā sintakse. Tās pārskatot, spēle strādāja un sapratu, ko mācīties talāk.

Izveidoto funkciju saraksts:

1. Dot function

```
def dot(X, Y):
    is_transposed = False

    X = np.atleast_2d(X)
    Y = np.atleast_2d(Y)

    if X.shape[1] != Y.shape[0]:
        is_transposed = True
        Y = np.transpose(Y)

    X_rows = X.shape[0]
    Y_columns = Y.shape[1]

    for X_row in range(X_rows):
        for Y_column in range(Y_columns):
            product[X_row, Y_column] = np.sum(X[X_row,:] * Y[:, Y_column])

    if is_transposed:
        product = np.transpose(product)

    if product.shape[0] == 1:
        product = product.flatten()

    return product
```

2. Vector normalization

```
def l2_normalize_vec2d(vec2d):
    length = math.sqrt(vec2d[0]**2 + vec2d[1]**2)
    normalized_vec2d = np.array([vec2d[0]/length, vec2d[1]/length])
    return normalized_vec2d
```

3. Translaiton matrix

```
def translation_mat(dx, dy):
    T = np.array([
        [1.0, 0.0, dx],
        [0.0, 1.0, dy],
        [0.0, 0.0, 1.0]
    ])
    return T
```

4. Scaling matrix

```
def scale_mat(dx, dy):
    T = np.array([
        [dx, 0.0, 0.0],
        [0.0, dy, 0.0],
        [0.0, 0.0, 1.0]
    ])
    return T
```

5. Circle generation

```
def drawCircle(radius):
        detail = 24
        circle = []
        d = 0
        x = 0
        while d < 375:
            circle.append([radius*np.cos(np.radians(d)), radius*np.sin(np.radians(d))])
            d +=375/detail
            x +=1

        return np.array(circle)
```

5. Additions

Pavadot laiku ar spēli mazliet vairāk, pievienoju ekstra elementus spēlei, lai to padarītu jautrāku un vizuāli pievilcīgāku:

5.0 Izveidoju emission particle objektu

```
class EmissionParticle(MovableObject):
    def __init__(self, directionVector, position):
        super().__init__()
        self.speed = .75
        I = np.array([
            [1, 0],
            [0, 1],
        ])

        self.vec_pos = dot(position, I)

        radius = np.random.uniform(0.15, 0.3)

        s = drawCircle(radius)
        self.geometry = s

        directionChangeMatrix = np.array([
            [np.random.uniform(-1.5, -0.5), 0],
            [0, np.random.uniform(-1.5, -0.5)],
        ])
        self.vec_dir = dot(directionVector, directionChangeMatrix)
        self.lifespan = 1
    def update_movement(self, dt):
        self.lifespan -= dt
        super().update_movement(dt)
        self.geometry = self.geometry * .75
        self.speed -= dt * 0.6
        if self.lifespan < 0:
            self.geometry = clearMatrix(self.geometry)


def createEmissionParticles(player):
```

```
    particles = []
    particles.append(EmissionParticle(player.vec_dir, player.vec_pos))
    particles.append(EmissionParticle(player.vec_dir, player.vec_pos))
    particles.append(EmissionParticle(player.vec_dir, player.vec_pos))
    return np.array(particles)
```

5.1 Pievienoju stratēģiju spēles izbeigšanai - ja planēta pietuvojas pārāk tuvu spēlētājam, spēle beidzas:

5.1.1 noteikt distanci starp diviem objektiem

```
def distanceBetweenTwoObjects(pos1, pos2):
    return np.sum((pos1 - pos2)**2)/2
5.1.2 Kā updatot izraisīto spēku spēlētājam, planētai pietuvojoties tuvāk

def updateForceOnPlayer(self:MovableObject):
    F = 9.82 * self.radius / distanceBetweenTwoObjects(self.vec_pos, player.vec_pos)*2
    F_vec = l2_normalize_vec2d(self.vec_pos - player.vec_pos)
    player.external_forces[self.planetNumber] = F * F_vec
```

5.1.3 Parbaude vai objekti ir saskrējušies:

```
def isCollided(firstObject:MovableObject, secondObject:MovableObject):
    d_2 = distanceBetweenTwoObjects(firstObject.vec_pos, secondObject.vec_pos)
    if d_2 < 0.2:
        return True
    return False
```

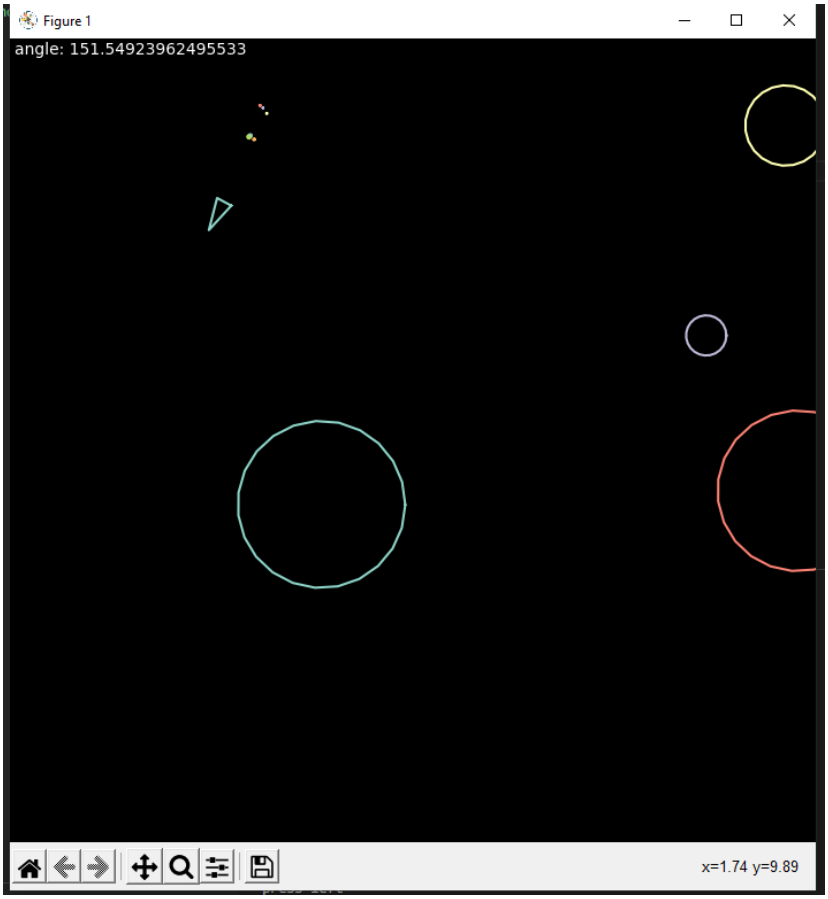
5.1.4 Spēles izbeigšana:

```
def closeWithGameOver():
    plt.text(x=-SPACE_SIZE+9, y=SPACE_SIZE-9, s=f'GAME OVER')
    plt.pause(5)
    global is_running
    is_running = False
```

5.1.5 Planētas update_movement implementācijai pievienoju izveidotās funkcijas:

```
class Planet(MovableObject):
    def __init__(self, name, index, radius):
        super().__init__()
        self.attribute_name = name
        self.speed = 0
        self.planetNumber = index
        print(radius)
        self.radius = radius

        s = drawCircle(self.radius)
        self.geometry = s

        self.vec_pos = np.array([np.random.uniform(-10.0, 10.0), np.random.uniform(-10.0, 10.0)])
        self.speed = 0

    def update_movement(self, dt):
        if isCollided(self, player):
            closeWithGameOver()

        super().update_movement(dt)

        updateForceOnPlayer(self)
```

Gala rezultātā tika izveidota programma, kas ļauj cilvēkam braukt pa vizuālu plakni ar trīsstūra figūru, šai figūrai ir jāizvairās no automātiski uzģenerētam planētām, kas iesūc figūru iekšā sevī, līdzko ta pietuvojās pārāk ātri. Figūrai ir iespējams paātrināt kustību, mainīt virzienu un lidot caur kartes sienām, lai izvairītos no planētām.



Attēlā redzama spēles gaita.

2.1.2. Inversā kinemātika

Šī uzdevuma mērķis bija izprast kā izmantot absolūto un kvadrātisko kļūdu algoritmos, lai nonāktu pie optimālā rezultāta. Lai sasniegtu mērķi, tiek izveidota programma, kura ar trīs savienotiem posmiem jeb "rokām" mēģina nokļūt pie noteikta galapunkta, koriģējot tās savienojumu leņķus. Šīs programmas izstrādes procesā tika atkārtots, kā ar programatūru ievietot funkciju atvasinājumus, kā arī kā sastādīt atsevišķus algoritma soļus, lai nonāktu pie optimālāka risinājuma garākā algoritmā.

Izveidoto funkciju saraksts:

1. Rotation matrix

```
def rotation(theta):
    cos_theta = math.cos(theta)
    sin_theta = math.sin(theta)
    return np.array([
        [cos_theta, -sin_theta],
        [sin_theta, cos_theta],
    ])
```

2. Derivative of rotation matrix

```
def d_rotation(theta):
    cos_theta = math.cos(theta)
    sin_theta = math.sin(theta)
    return np.array([
```

```
        [-sin_theta, -cos_theta],
        [cos_theta, -sin_theta],
    ])
```

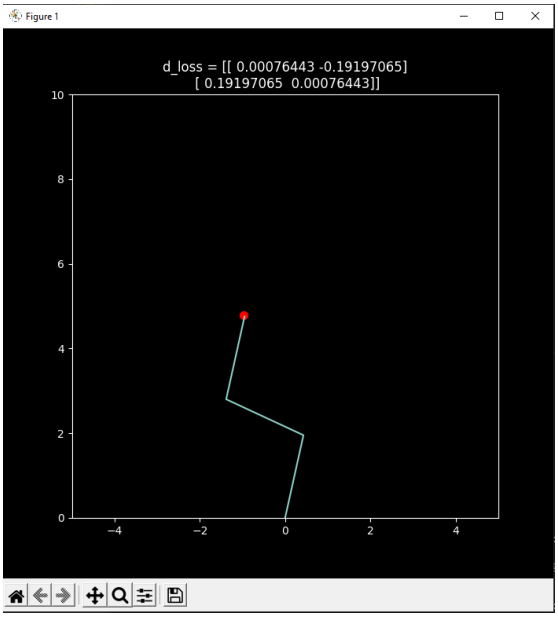
3. Angle rotation algorithm

```
d_theta_1 = np.sum(2 * (point_3 - target_point) * (dR1 @ t + dR1 @ R2 @ t))
theta_1 -= d_theta_1 * alpha

d_theta_2 = np.sum(2 * (point_3 - target_point) * (R1 @ dR2 @ t))
theta_2 -= d_theta_2 * alpha

d_theta_3 = np.sum(2 * (point_3 - target_point) * (R2 @ dR3 @ t)  )
theta_3 -= d_theta_3 * alpha
```

4. Derivative of mse loss function:

```
d_loss = 2*np.mean(target_point - point_3)
```

Gala rezultātā tiek izveidota robota roka, kas tuvusies katram punktam, ko lietotājs būs nospiedis uz ekrāna.



Attēls ar roku pietuvojušos atzīmētajam punktam

2.1.3. Vidējās kvadrātiskās kļūdas un vidējās absolūtās kļūdas salīdzinājums

Šī uzdevuma mērķis bija salīdzināt un izprast, kad labāk lietot vidējo absolūto kļūdu un kad – vidējo kvadrātisko kļūdu. Uzdevuma beigās tiek secināts, ka vispārējos terminos vidējā absolūtā kļūda ir piemērotāka datu kopām, kurām ir mazāk datu izņēmumu t.i. datu kuri drastiski atsķīras no pārējiem, jo vidējā kvadrātiska kļūda spēcīgi izceļ izņēmumu datus ierakstus.

Izveidoto funkciju saraksts:

1. Sigmoid function

```
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))
```

2. Loss mae

```
def loss_mae(y_prim, y):
    return np.sum(np.abs(y_prim - y))
```

3. Loss mse

```
def loss_mse(y_prim, y):
    return np.mean(np.sum((y_prim - y)**2))
```

4. Model

```
def model(x, W_1, b_1, W_2, b_2):
    layer_1 = linear(W_1, b_1, x)
    layer_2 = sigmoid(layer_1)
    layer_3 = linear(W_2, b_2, layer_2)
    return layer_3
```

2.1.4. Lineārās regresijas uzdevums ar apmācības modeli

Šī uzdevuma mērķis bija konstruēt neironu tīklu, kas funkcionēs ar noteiktu vairākslāņu algoritmisko modeli, lai izrēķinātu lineārās regresijas problēmu. Šī uzdevuma ietvaros, tika izstrādāts modelis, kas paredz noteiktu iznākumu funkcijai, balstoties uz ievades datiem. Sākuma modelis tiek izstrādāts, lai tas varētu ieņemt viena elementa datus (f(x)), bet pēctam, tas tiek uzlabots, lai iekļautu vairāku dimensiju datus (f(x, z)). Šī uzdevuma izpildes laikā, tika atkārtoti daudzi svarīgi koncepti, kas tiks pielietoti bakalaura izstrādes laikā, ka piemēram – neironu tīkla modeļa izveide, modeļa atpakaļizplatīšanās, stohastiskā gradienta nolaišanās u.c. Šī uzdevuma laikā, saskāros ar daudz problēmām, kuras darba vadītājs man izskaidrojot deva skaidrāku sapratni par to ka mašīnapmācības modeļu iekšējie tīkli strādā un kā tos matemātiski izprast.

Izveidoto funkciju saraksts:

1. Linear function

```
def linear(W, b, x):
    prod_W = np.squeeze(W.T @np.expand_dims(x, axis=-1), axis=-1)
    return prod_W + b
```

2. Derivatives for each variable

```
def dW_linear(W, b, x):
    return x

def db_linear(W, b, x):
    return 1

def dx_linear(W, b, x):
    return W
```

3. Back propogation

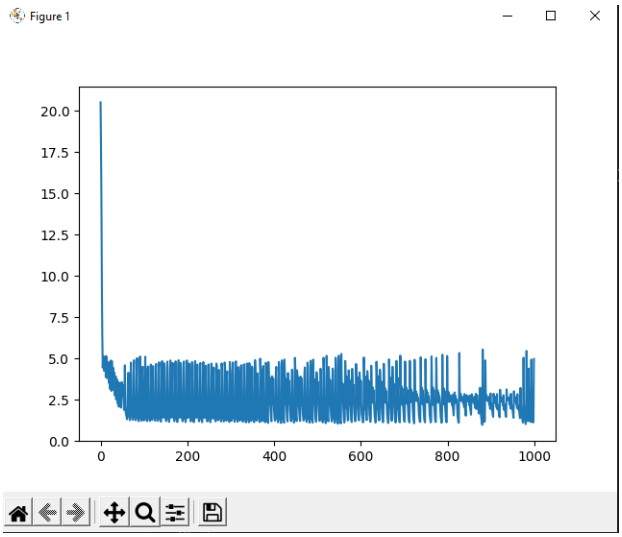
```
def dy_prim_loss_mae(y_prim, y):
    return (y_prim - y) / (np.abs(y_prim - y) + 1e-8)

def dW_1_loss(x, W_1, b_1, W_2, b_2, y_prim, y):
    d_layer_1 = dW_linear(W_1, b_1, x)
    d_layer_2 = dx_sigmoid(linear(W_1, b_1, x))
    d_layer_3 = np.expand_dims(dx_linear(W_2, b_2, sigmoid(linear(W_1, b_1, x))), axis=-1)
    d_loss = dy_prim_loss_mae(y_prim, y)
    d_dot_3 = np.squeeze(d_loss @ d_layer_3, axis=-1).T
    return d_dot_3 * d_layer_2 * d_layer_1

def db_1_loss(x, W_1, b_1, W_2, b_2, y_prim, y):
    d_layer_1 = db_linear(W_1, b_1, x)
    d_layer_2 = dx_sigmoid(linear(W_1, b_1, x))
    d_layer_3 = np.expand_dims(dx_linear(W_2, b_2, sigmoid(linear(W_1, b_1, x))), axis=-1)
```

```
    d_loss = dy_prim_loss_mae(y_prim, y)
    d_dot_3 = np.squeeze(d_loss @ d_layer_3, axis=-1).T
    return d_dot_3 * d_layer_2 * d_layer_1

def dW_2_loss(x, W_1, b_1, W_2, b_2, y_prim, y):
    d_layer_3 = dW_linear(W_2, b_2, sigmoid(linear(W_1, b_1, x)))
    d_loss = dy_prim_loss_mae(y_prim, y)
    return d_loss * d_layer_3

def db_2_loss(x, W_1, b_1, W_2, b_2, y_prim, y):
    d_layer_3 = db_linear(W_2, b_2, sigmoid(linear(W_1, b_1, x)))
    d_loss = dy_prim_loss_mae(y_prim, y)
    return d_loss * d_layer_3
```

4. SGD implementation

```
dW_1 = np.sum(dW_1_loss(X, W_1, b_1, W_2, b_2, Y_prim, Y))
    dW_2 = np.sum(dW_2_loss(X, W_1, b_1, W_2, b_2, Y_prim, Y))
    db_1 = np.sum(db_1_loss(X, W_1, b_1, W_2, b_2, Y_prim, Y))
    db_2 = np.sum(db_2_loss(X, W_1, b_1, W_2, b_2, Y_prim, Y))

    W_1 -= dW_1 * learning_rate
    W_2 -= dW_2 * learning_rate
    b_1 -= db_1 * learning_rate
    b_2 -= db_2 * learning_rate
```

Rezultātā tiek iegūts grafiks, kas atspoguļo funkcijas rezultātu sadalījumu vērtībām no 0 līdz 1000. Te jaņem vērā ka modelis tika trenēts ar mazu ievaddatu daudzumu, tāpēv rezultāti nav tik ļoti noderīgi ka informatīvi.



2.1.5. Apmācības modelis mašīnu cenu pareģošanai

Šajā uzdevumā tika konstruēts dziļās mašīnapmācības modelis, lai varētu paredzēt mašīnu cenas izmantojot Indijas sludinājumu portāla datu setu. Lai īstenotu mērķi, tika implementētas MSE un NRMSE kļūdas aprēķina funkcijas kā arī ReLU un Swish aktivizācijas funkcijas.

Izveidoto funkciju saraksts:

1. MSE/L2 funkcija

```
class LossMSE():
    def __init__(self):
        self.y = None
        self.y_prim  = None

    def forward(self, y: Variable, y_prim: Variable):
        self.y = y
        self.y_prim = y_prim
        loss = np.mean(np.sum((y.value - y_prim.value)**2))
        return loss

    def backward(self):
        self.y_prim.grad += -2*(self.y.value - self.y_prim.value)
```

2. Relu funkcija

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

    def forward(self, x: Variable):
        self.x = x
        temp = self.x.value
        temp[temp<0]=0
        self.output = Variable( temp )
        return self.output

    def backward(self):
        temp = self.output.value
        temp[temp<0]=0
        temp[temp>0]=1
        self.x.grad += temp * self.output.grad
```

3. NRMSE funkcija

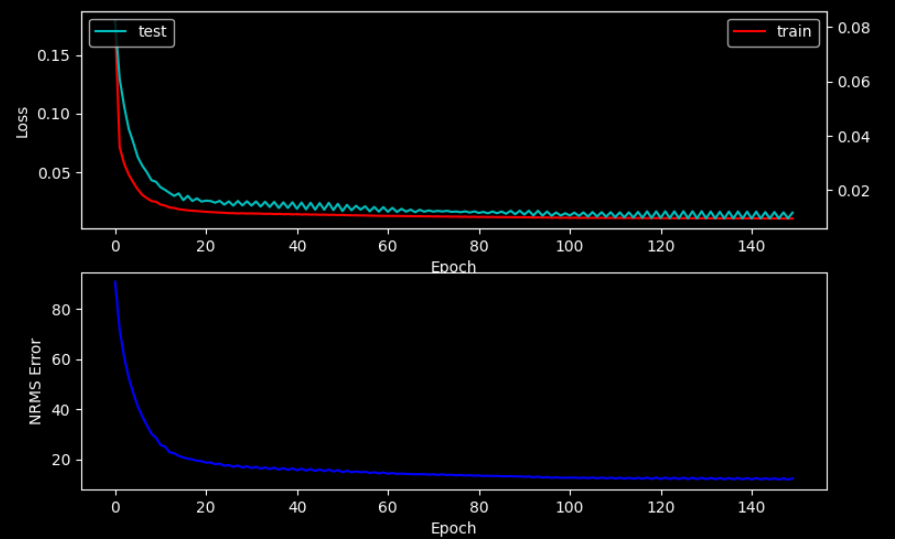
```
def calculateNRMSE(y, y_prim):
    rmse = np.sqrt(np.mean(np.sum((y_prim - y)**2)))
    result = rmse/np.std(y)
```

```
        return result
```

4. Swish funkcija

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

        def forward(self, x: Variable):
            self.x = x
            self.output = Variable(x.value / (1.0 + np.exp(-x.value)))
            return self.output

        def backward(self):
            self.x.grad += self.output.value + np.std(x) * (1.0 - self.output.value)
```

```
def backward(self):
    self.y_prim.grad += -(self.y.value - self.y_prim

lass Model:
  def __init__(self):
    self.layers = [
        LayerLinear(in_features=6, out_features=8),
        LayerRelu(),
        LayerLinear(in_features=8, out_features=12),
        LayerRelu(),
        LayerLinear(in_features=12, out_features=7),
        LayerRelu(),
        LayerLinear(in_features=7, out_features=2),
    ]

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

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

  def parameters(self):
    variables = []
    for layer in self.layers:
```



2.1.6. Pytorch bibliotēkas izmantošana regresijas modelim

Šajā uzdevumā mērķis bija iepazīties ar Pytorch bibliotēku un izprast tās lietojumu dziļo mašīnapmācības modeļu izveidē. Uzdevuma laikā tika apskatīts kā mašīnapmācības modelī strādāt ar datiem, kas satur nelineārus ievaddatus, t.i. kategorizētus ievaddatus – mašīnu dzinēja tips, marka, ražotājs utt. Darba gaitā tika apgūts Huber loss funkcijas pielietojums.

Izveidoto funkciju saraksts:

1. Embedding matricas slāņa apstrāde

```
elf.embs = torch.nn.ModuleList()
        for i in range(4): # brand, fuel, transmission, dealership
            self.embs.append(
                torch.nn.Embedding(
                    num_embeddings=len(dataset_full.labels[i]),
                    embedding_dim=3
```

```
                )
            )

    def forward(self, x, x_classes):
        x_emb_list = []
        for i, emb in enumerate(self.embs):
            x_emb_list.append(
                emb.forward(x_classes[:, i])
            )
        x_emb = torch.cat(x_emb_list, dim=-1)
        x_cat = torch.cat([x, x_emb], dim=-1)
        y_prim = self.layers.forward(x_cat)
        return y_prim
```

2. Huber Loss

```
class LossHuber(torch.nn.Module):
    def __init__(self, delta):
        super().__init__()
        self.delta = delta

    def forward(self, y_prim, y):
        return torch.mean(self.delta**2 * (torch.sqrt(1 + ((y - y_prim)/self.delta) ** 2) - 1))
```

2.1.7. Numpy bibliotēkas izmantošana klasifikācijas uzdevumam

Šajā darbā tika apgūtas dažādas funkcijas mašīnapmācības modeļa izveidei izmantojot numpy bibliotēku. Funkcijas kas tika apgūtas – Softmax aktivizācijas funkcija un “Cross-entropy loss” funkcija modeļa svaru izmaiņai.

Izveidoto funkciju saraksts:

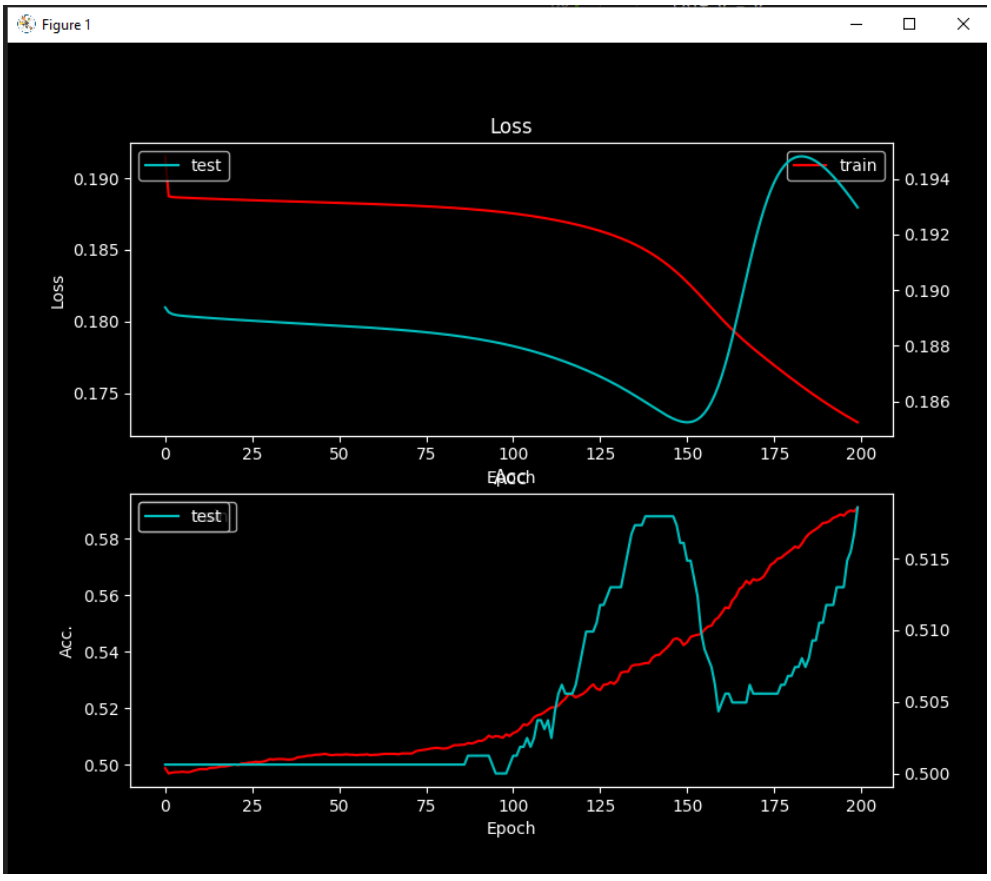
1. Softmax

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

    def forward(self, x):
        self.x = x

        np_x = np.array(x.value)
        np_x -= np.max(np_x, axis=-1, keepdims=True)
        np_e_x = np.exp(np_x)

        self.output = Variable(
            np_e_x / np.sum(np_e_x, axis=-1, keepdims=True)  # var arī [:, np.newaxis]
        )
        return self.output

    def backward(self):
```

```
        size = self.x.value.shape[-1]
        J = np.zeros((BATCH_SIZE, size, size))
        a = self.output.value

        for row in range(size):
            for column in range(size):
                if row == column:
                    J[:, row, column] = a[:,row] * (1 - a[:, column])
                else:
                    J[:, row, column] = -a[:,row] * a[:, column]

        self.x.grad += np.squeeze(J @ np.expand_dims(self.output.grad, axis=-1), axis=-1)
```

2. Cross-entropy Loss

```
class LossCrossEntropy():
    def __init__(self):
        self.y_prim = None

    def forward(self, y, y_prim):
        self.y = y
        self.y_prim = y_prim
        return np.mean(-y.value * np.log(y_prim.value + 1e-8))

    def backward(self):
         self.y_prim.grad = -self.y.value / (self.y_prim.value + 1e-8)
```



2.1.8. Konvolūcijas tīkla implementācija

Šajā darbā tika īstenots konvolūcionāls neironu tīkls, lai klasificētu vizuālus attēlus izmantojot "Fashion-MNIST" datu kopu. Šī uzdevuma ietvaros tika apskatīts, kā neironu tīkli apstrādā vizuālas bildes un kādas metodes tiek pielietotas, lai labāk apmācītu un novērtētu modeli, kas spējīgs kategorizēt vizuālus attēlus. Šajā darbā arī tika apskatīts, kā izvēlēties izmantotās ierīces resursus, lai veiktu modeļa izpildi uz videokartes nevis CPU.

Izveidoto funkciju saraksts:

1. Conv2d

```
def forward(self, x):
        batch_size = x.size(0)
        in_size = x.size(-1)
        out_size = get_out_size(in_size, self.padding, self.kernel_size, self.stride)

        out = torch.zeros(batch_size, self.out_channels, out_size, out_size)

        x_padded_size = in_size + self.padding*2
        if self.padding:
            x_padded = torch.zeros(batch_size, self.in_channels, x_padded_size, x_padded_size)
            x_padded[:, :, self.padding:-self.padding, self.padding:-self.padding] = x
        else:
            x_padded = x.to(DEVICE)

        K = self.K.view(-1, self.out_channels) # kernel_size8kernel_size*in_channels

        i_out = 0
        for i in range(0, x_padded_size-self.kernel_size+1, self.stride):
            j_out = 0
            for j in range(0, x_padded_size-self.kernel_size+1, self.stride):
                x_part = x_padded[:, :, i:i+self.kernel_size, j:j+self.kernel_size]
                x_part = x_part.reshape(batch_size, -1)

                out_part = x_part @ K # (batch, out_channels)
                out[:, :, i_out, j_out] = out_part

                j_out += 1
            i_out += 1

        return out
```

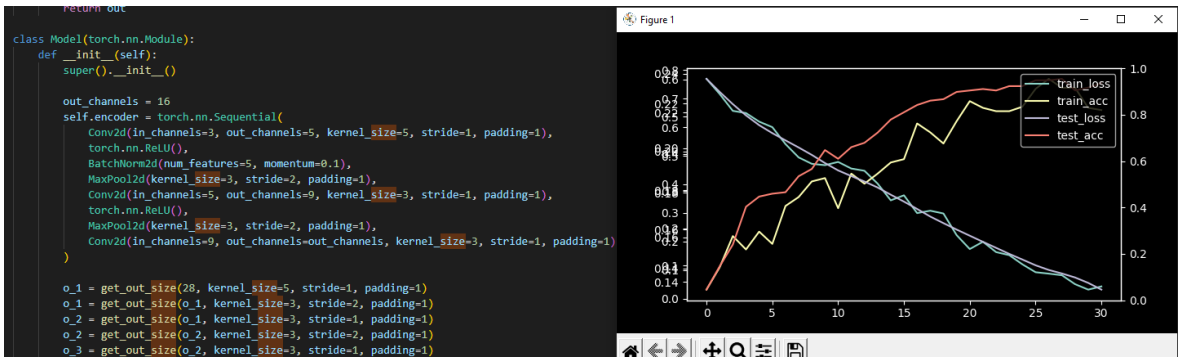
2. MaxPool2d

```
def forward(self, x):
        batch_size = x.size(0)
        channels = x.size(1)
        in_size = x.size(-1)
        out_size = get_out_size(in_size, self.padding, self.kernel_size, self.stride)

        out = torch.zeros(batch_size, channels, out_size, out_size).to(DEVICE)
```

```
            x_padded_size = in_size + self.padding * 2
            if self.padding:
                x_padded = torch.zeros(batch_size, channels, x_padded_size, x_padded_size).to(DEVICE)
                x_padded[:, :, self.padding:-self.padding, self.padding:-self.padding] = x
            else:
                x_padded = x.to(DEVICE)

            i_out = 0
            for i in range(0, x_padded_size-self.kernel_size+1, self.stride):
                j_out = 0
                for j in range(0, x_padded_size-self.kernel_size+1, self.stride):
                    x_part = x_padded[:, :, i:i+self.kernel_size, j:j+self.kernel_size]
                    x_part = x_part.reshape(batch_size, channels, -1)

                    out_part = torch.max(x_part, dim=-1).values # x_part @ K # (batch, out_channels)
                    out[:, :, i_out, j_out] = out_part

                    j_out += 1
                i_out += 1

            return out
```

3. BatchNorm2d

```
def forward(self, x):
        if x.size(1) != self.num_features:
            raise Exception('Wrong channel count in batchnorm')

        if self.moving_mean.device != x.device:
            self.moving_mean = self.moving_mean.to(x.device)
            self.moving_var = self.moving_var.to(x.device)

        if self.training:
            mean = x.mean(dim=(0,2,3), keepdims=True)
            var = ((x - mean)**2).mean(dim=(0,2,3), keepdims=True)

            self.moving = mean * self.momentum + self.moving_mean * (1 - self.momentum)
            self.moving_var = var * self.momentum + self.moving_var * (1 - self.momentum)
            out_norm = (x - mean) / torch.sqrt(var + 1e-5)
        else:
            out_norm = (x - self.moving_mean) / torch.sqrt(self.moving_var + 1e-5)
        out = self.gamma * out_norm + self.beta
        return out
```



2.1.9. “Autoencoder” un “Denoising autoencoder” neironu tīklu modeļi

Šajā darbā tika turpināta iepriekšējā darbā iesāktā vizuālu attēlu analīzes algoritmu izveide. Tika apskatīts kā strādā autoencoder un denoising eutoencoder neironu modeļi un tie tikai izveidoti, kā rezultātā tika ģenerēti attēlu atveidojumi, ko uzģenerēja apmācītais modelis. Tika apskatīts, kā uzlabot tīkla stabilitāti ar trokšņa palielināšanu ievaddatu kopā.

Izveidoto funkciju saraksts:

1. Encoder

```
self.encoder = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels=3, out_channels=4, kernel_size=7, stride=3, padding=0),
            torch.nn.BatchNorm2d(num_features=4),
            torch.nn.LeakyReLU(),

            torch.nn.Conv2d(in_channels=4, out_channels=8, kernel_size=8, stride=4, padding=2),
            torch.nn.BatchNorm2d(num_features=8),
            torch.nn.LeakyReLU(),

            torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=4, stride=2, padding=1),
            torch.nn.BatchNorm2d(num_features=16),
            torch.nn.LeakyReLU(),

            torch.nn.Conv2d(in_channels=16, out_channels=32, kernel_size=4, stride=2, padding=1),
            torch.nn.BatchNorm2d(num_features=32),
            torch.nn.LeakyReLU(),

            torch.nn.Conv2d(in_channels=32, out_channels=32, kernel_size=4, stride=2, padding=1),
            torch.nn.BatchNorm2d(num_features=32),
            torch.nn.Tanh()
)
```

2. Decoder

```
self.decoder = torch.nn.Sequential(
            torch.nn.ConvTranspose2d(in_channels=32, out_channels=24, kernel_size=4, stride=2, padding=1),
            torch.nn.BatchNorm2d(num_features=24),
            torch.nn.LeakyReLU(),

            torch.nn.ConvTranspose2d(in_channels=24, out_channels=12, kernel_size=4, stride=2, padding=1),
```

```
            torch.nn.BatchNorm2d(num_features=12),
            torch.nn.LeakyReLU(),

            torch.nn.ConvTranspose2d(in_channels=12, out_channels=8, kernel_size=4, stride=2, padding=1),
            torch.nn.BatchNorm2d(num_features=8),
            torch.nn.LeakyReLU(),

            torch.nn.ConvTranspose2d(in_channels=8, out_channels=4, kernel_size=8, stride=4, padding=2),
            torch.nn.BatchNorm2d(num_features=4),
            torch.nn.LeakyReLU(),

            torch.nn.ConvTranspose2d(in_channels=4, out_channels=4, kernel_size=7, stride=3, padding=0),
            torch.nn.BatchNorm2d(num_features=4),
            torch.nn.LeakyReLU(),

            torch.nn.ConvTranspose2d(in_channels=4, out_channels=3, kernel_size=3, stride=1, padding=1),
            torch.nn.BatchNorm2d(num_features=3),
            torch.nn.Sigmoid()
        )
```

3. Trokšņa pievienošana

```
    def applyNoise(self, x):
        if np.random.random() < 0.5:
            noise = torch.randn(x.size())
            x[noise < 0.5] = 0
```

2.2. Analizētie rakstu avoti

Raksti, kas tika analizēti kopumā bija ļoti vispārīgi, primāri fokusēti par Beiesa neirālajiem tīkliem un tiem netika veltīts tik daudz laiks, cik tika veltīts prieks praktiskajiem uzdevumiem.

2.2.1. Monte Carlo Dropout

Michał Oleszak

Monte Carlo Dropout (2020) [tiešsaiste].

Pieejams: https://towardsdatascience.com/monte-carlo-dropout-7fd52f8b6571

Raksts apraksta ‘Monte Carlo Dropout’ metodi, kas tiek izmantota, lai izvairītos no pārmērīgas pielāgošanas pie treniņa datiem. Izplatīta problēma mašīnmācīšanās algoritmiem notiek, kad algoritms tiekot trenēts, pārlieki pieskaņojas trennēšanas datiem. Šo problēmu angliski sauc par ‘Overfitting’ jeb

pārapmācīšanās. Izvairīties no šīs problēmas var izmantojot ‘Monte Carlo Dropout’ metodi. Šī metode darbojas pēc principa, ka neironu tīklā, kamēr modelis tiek trenēts, atsevišķi neironi tiek ignorēti – iespējamību kādam neironam tikt ignorētam ir kāda iespējamības vērtība no 0 līdz 1, ko dēvē par ‘Dropout rate’ jeb caurkrites funkcija. Konceptuāli, šī metode palīdz modelim izvairīties no pārapmācīšanās (Overfitting) neļaujot modelim pārlieki paļauties uz nevienu ievaddatu, tādējādi izdalot modeļa svarus vienmērīgāk caur neironu tīklu, kad modelis tiek pielietots, visi neironi tiek ieslēgti un neironu ignorēšana nenotiek.

Raksts tālāk apskata gadījumu, kurā pielietojot ‘Monte Carlo Dropout’ metodi tiek iegūti par 15% mazāk kļūdainu rezultātu, nemainot neko citu modelī vai izmantotajās datu kopās – skaidri demonstrējot šīs metodes efektīvumu. Raksta autors arī paskaidro, ka šo metodi var interpretēt kā Beiesa tuvinājumu – izveidojot vairākus neironu tīklus (katrs ar atsevišķiem izslēgtiem neironiem) tiek iegūta grupa ar paraugiem, kuru kopums izveido vislabāko neironu tīkla svaru vērtības.

2.2.2. Why you should use Bayesian Neural Network

Yeung Wong
Why you should use Bayesian Neural Network? (2021) [tiešsaiste].
Pieejams: https://towardsdatascience.com/why-you-should-use-bayesian-neural-network-aaf76732c150

Raksts izskaidro kas ir Beiesa neironu tīkli un salīdzina tos ar citiem neironu tīkliem. Autors, vienkāršojot skaidro, ka Beiesa neironu tīkli izmanto vērtību sadalījumus konstantu vērtību vietā neironu tīkla svariem un rezultātiem. Faktiski, neironu tīklos, vērtības no viena slāņa uz nākošo tiek padotas aprēķinot ievaddatu vērtības ar svaru reizinājumu summu, bet Beiesa neirālajos tīklos, svari ir vērtību sadalījumi, tāpēc, neironu rezultāti izveido vērtību sadalījumus, nevis konstantas vērtības.

Raksts tiek apskatīti pozitīvie un negatīvie Beiesa neironu tīklu aspekti. Kā pozitīvos aspektus autors min spēcīgāku modeļa izveidi un rezultāta automātisko nenoteiktības vērtību t.i. ja klasisks neirālais tīkls izveidos rezultātu kā skaitli, tad Beiesa neirālais tīkls izdos vērtību sadalījumu, tādā veidā izveidojot lielāku sapratni par rezultātu potenciālo kļūdu. Kā negatīvos aspektus Beiesa neironu tīkliem autors izceļ matemātisko sarežģītību, kas samazina potenciālo praktikantu skaitu un ilgāku trenēšanas laiku modelim. Ilgāks trenēšanas laiks, var tikt izskaidrots ar papildu darbībām, kas jāveic modelim, lai aprēķinātu visus vērtību sadalījumus, kas nav nepieciešams citiem neironu tīkliem.

IZMANTOTIE INFORMĀCIJAS AVOTI

Machine Learning as a Service Market Size, Report 2030 (2022) [tiešsaiste].
Precedence Research, https://www.precedenceresearch.com [2022]
Pieejams: https://www.precedenceresearch.com/machine-learning-as-a-service-market

Michał Oleszak
Monte Carlo Dropout (2020) [tiešsaiste].
Pieejams: https://towardsdatascience.com/monte-carlo-dropout-7fd52f8b6571

Yeung Wong
Why you should use Bayesian Neural Network? (2021) [tiešsaiste].
Pieejams: https://towardsdatascience.com/why-you-should-use-bayesian-neural-network-aaf76732c150