Deep Learning for Natural Language Processing

Elias Wendt, 2754066
Laura Boyette, 2271837

TECHNISCHE UNIVERSITÄT DARMSTADT

Summer Term 2022
Sheet 1

Task 1.1: Setup

Task 1.2: Sigmoid Activation Function

sig(x) $= \frac{1}{1+e^{-x}}$

Show that: $sig'(x) = sig(x) * (1 - sig(x))$

$$= \frac{1}{1+e^{-x}} * (1 - \frac{1}{1+e^{-x}})$$

$$= \frac{1}{1-e^{-x}} - \frac{1}{(1+e^{-x})^2}$$

$$= \frac{1+e^{-x}}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2}$$

$$= \frac{e^{-x}}{(1+e^{-x})^2}$$

Task 1.3: Tensorflow Playground

1.3a) Circular Dataset

A perceptron architecture can learn a good discriminator, if we use squared input activation for both axes.

1.3b) Multi Layer Perceptron

The best scenario was with the most neurons in the first hidden layer, and fewer neurons in the final hidden layer. This is also the fastest converging scenario, because the NN can consider all of the various input activations. Towards the last layer it condenses the information into fewer dimensions. Having the same number of neurons throughout all layers should provide it with the same power, but has more parameters and is therefore slower to train.

Task 1.4: Softmax

Softmax provides a so-called `multinomial` probability distribution. This is required in non-binary classification problems. Where conventional activation functions simply map a single input value to a probability, Softmax sets the values of multiple inputs in relation and calculates the probabilites of all outputs so that they add up to 1.

Task 1.5: Sentiment Polarity in Movie Reviews

1.5a) Dataset Reader

1.5b) Numpy Implementation

```
import numpy as np
#from tqdm import tqdm

# ------------------------------------------------------------
#                 5.1 DATASET READER
# ------------------------------------------------------------
def read_dataset(filepath: str):

    datapoints = []

    with open(filepath) as f:
        for line in f:
            _, sentiment_str, embedding_str = line.split("\t")

            y = np.array(1.0 if sentiment_str == "label=POS" else 0.0)

            embedding = [x for x in embedding_str.split(" ")]
            embedding.append(1.0)
            x = np.array(embedding, dtype=np.double)

            datapoints.append((x, y))

    return datapoints

# ------------------------------------------------------------
#                 5.2 NUMPY IMPLEMENTATION
# ------------------------------------------------------------

def sig(x):
    return 1 / (1 + np.exp(-x))


def sig_deriv(x):
    return sig(x)*(1-sig(x))


def loss(y, y_hat):
    return (y_hat - y)**2


def square_loss(datapoints, w):
    sum = 0

    for x, y in datapoints:
        sum += (forward(x, w) - y) ** 2

    return sum
```

```
def accuracy(datapoints, w):
    tp = 0
    for x, y in datapoints:
        predicted = np.round(forward(x, w))

        if predicted == y:
            tp += 1

    return tp / len(datapoints)

# ------------------------------------------------------------------
#                 5.3  TRAINING
# ------------------------------------------------------------------

def forward(x, w):
    return sig(x.T.dot(w))


def backward(w, lr, batch):

    sum = np.zeros(w.shape)

    for x, y in batch:

        # print(f"x.shape={x.shape}, y.shape={y.shape}, w.shape={w.shape}")

        sum += (forward(x, w) - y) * sig_deriv(x.T.dot(w)) * x

    # print(f"sum.shape={sum.shape}, w.shape={w.shape}")

    w -= (lr / len(batch)) * sum


def train():
    # initialize weights randomly (with seed)
    np.random.seed(100)


    datapoints_dev = read_dataset("DATA/rt-polarity.dev.vecs")
    datapoints_test = read_dataset("DATA/rt-polarity.test.vecs")
    datapoints_train = read_dataset("DATA/rt-polarity.train.vecs")

    datapoints = datapoints_train

    # organize input in batches
    # shuffle batches

    N = len(datapoints[0][0]) # length of embedding vec

    # reshaping is very important here to allow multiplication in backward()
    w = np.random.normal(0, 1, (N, 1)).reshape(101,)

    lr = .01
    n_epochs = 300
```

```
    batch_size = 10

    n_batches = np.ceil(len(datapoints) / batch_size)

    for epoch in range(n_epochs):

        # shuffle batches before each epoch
        np.random.shuffle(datapoints)

        # form batches
        batches = np.array_split(datapoints, n_batches)

        for n_batch, batch in enumerate(batches):

            loss_sum = 0

            # iterate over all datapoints in this batch
            for x, y in batch:
                # forward
                y_hat = forward(x, w)

                loss_sum += loss(y, y_hat)

            #print(f"epoch={epoch}, n_batch={n_batch}, loss_sum={loss_sum}")

            # backward
            backward(w, lr, batch)

        # print matrics
        print("epoch={}/{}\n\tsquare_loss(dev)={:.2f}\n\tsquare_loss(test)={:.2f}\n\tacc(dev
            epoch+1, n_epochs,
            square_loss(datapoints_dev, w),
            square_loss(datapoints_test, w),
            accuracy(datapoints_dev, w),
            accuracy(datapoints_test, w)
        ))

if __name__ == "__main__":
    train()
```

1.5c) Training

With given Hyper Parameters:
`learning_rate = .01   n_epochs = 50   batch_size = 10`

Results (after 50 epochs):
```
square_loss(dev)=810.75
square_loss(test)=788.36
acc(dev)=0.49
acc(test)=0.51
```

With tuned Hyper Parameters
`learning_rate = .01   n_epochs = 300   batch_size = 10`

Results (after 300 epochs):
```
square_loss(dev)=451.15
square_loss(test)=468.36
acc(dev)=0.71
acc(test)=0.70
```