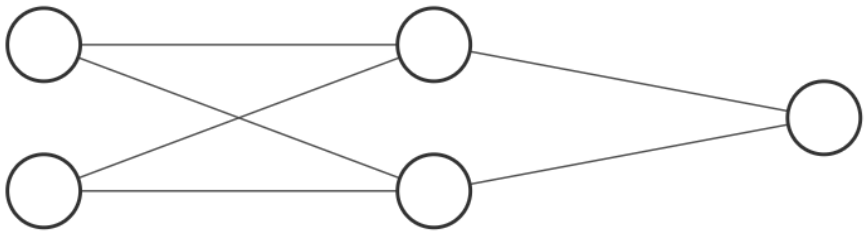
Division

Einem kleinen neuronalen Netz durch eine eigene Aktivierungsfunktion die Division beibringen



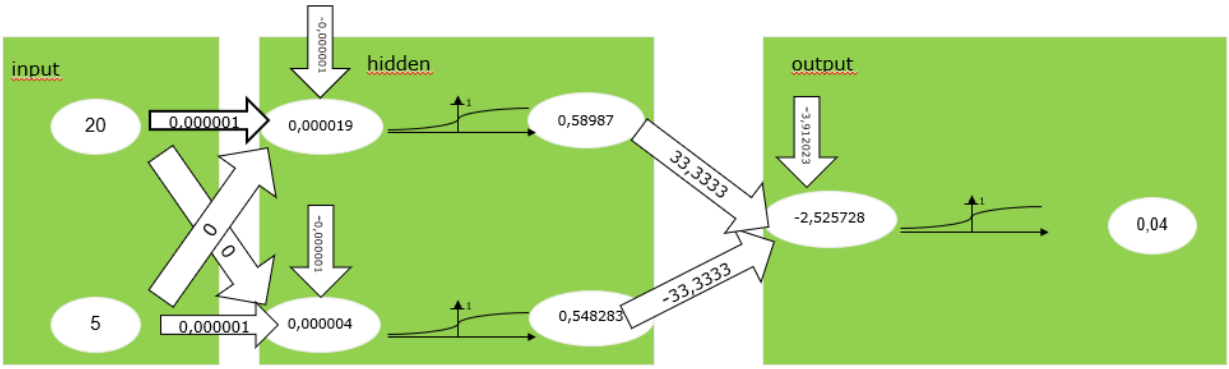
Input Layer $\in \mathbb{R}^2$ Hidden Layer $\in \mathbb{R}^2$ Output Layer $\in \mathbb{R}^1$

28. Juli 2021

Inhaltsverzeichnis

1. Theorie

Das Ziel des Projektes ist es, einem kleinen neuronalen Netz mit nur 2 Neuronen in der Zwischenschicht die Division beizubringen. Die Inputs sollen 2 positive Zahlen sein, und als Output soll das Ergebnis dividiert durch 100 herauskommen. Das Bild zeigt eine Möglichkeit von Gewichten und Biases, mit denen die Division funktioniert.



Da im Feed-forward-Vorgang eines neuronalen Netzes allerdings nur multipliziert und addiert wird, ist es mit einer standardisierten Aktivierungsfunktion nicht möglich, mit so wenigen Neuronen ein funktionierendes Divisionsmodell zu bauen. Was wir also brauchen, ist eine eigene Aktivierungsfunktion, die auf der einen Seite Exponentialrechnung nutzt und auf der anderen den Logarithmus.

$$y = 1{,}359140915\, e^{(x-1)} \qquad \text{for all x <= 0}$$
$$y = 0{,}03\, \ln(1000000x + 1) + 0{,}5 \qquad \text{for all x > 0 and x <= 15}$$

$$y = 1 - \frac{1}{109{,}0858178\, x - 1403{,}359435} \qquad \text{for all x > 15}$$

1.1. Python-Funktion

Als Erstes erstellen wir eine simple Python-Funktion, um die Aktivierungsfunktion abzubilden.

```python
import math

def divisionHard(i1, i2):
    h1 = AktivierungsFunktionHard(i1 * 0.000001 - 0.000001)
    h2 = AktivierungsFunktionHard(i2 * 0.000001 - 0.000001)
    output = AktivierungsFunktionHard(h1 * 33.3333 + h2 * -33.3333 - 3.912023)
    return output

def AktivierungsFunktionHard(x):
    if x <= 0:
        return 1.359140915 * math.exp(x - 1)
    elif x > 15:
        return 1 - 1/(109.0858178 * x - 1403.359435)
    else:
        return 0.03 * math.log(1000000 * x + 1) + 0.5
```

Testen wir die Funktion mit dem Beispiel aus dem Bild.

```
print(divisionHard(20, 5)) -> 0.03999994478806479
```

2. Keras

2.1. Sigmoid-Modell

Da wir uns jetzt der Theorie sicher sind, können wir als Nächstes das einfache neuronale Netz in Keras nachbauen. Da unser Output im Dezimalbereich liegt, können wir für den Anfang Sigmoid als Aktivierungsfunktion nutzen.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(2, activation='sigmoid'),
    Dense(1, activation='sigmoid'),
])
```

2.2. Trainingsdaten

Als Nächstes brauchen wir Trainingsdaten, die wir selbst in Python erstellen können.

```
def devision_data(size):
    xdata = []
    ydata = []
    for i in range(size):
        i1 = float(decimal.Decimal(random.randrange(100, 2000))/100)
        i2 = float(decimal.Decimal(random.randrange(100, 2000))/100)
        y = i1 / i2 / 100
        xdata.append([i1, i2])
        ydata.append([y])

    return np.array(xdata), np.array(ydata)

X_train, Y_train = devision_data(24000)
X_test, Y_test = devision_data(6000)
```

Nun können wir das Modell mit unseren eigens erstellten Daten trainieren lassen und evaluieren.

Für den Anfang wählen wir eine Batchsize von 32 und 90 Epochen. Als Optimizer nehmen wir Adam und als Kostenfunktion den MAE (mean absolute error), da dieser für uns am besten zu vergleichen ist.

```
model.compile(optimizer='adam',
              loss='mean_absolute_error')

hist = model.fit(X_train, Y_train,
          batch_size=32, epochs=90)

model.evaluate(X_test, Y_test)
```

Dieses Modell lernt zumindest und erzielt bessere Ergebnisse als durch strategisches Raten. Trotzdem ist es immer noch sehr stumpf.

2.3. Aktivierungsfunktion

Lasst uns also im nächsten Schritt unsere eigene Aktivierungsfunktion in Keras definieren. Die Funktionsabschnitte definieren wir über mehrere torch.where()-Verschachtelungen.

```
def custom_activation(x):
    smallerEqualZero = tf.less_equal(x, tf.constant(0.0))
    greaterZero = tf.greater(x, tf.constant(0.0))
    greaterFiveteen = tf.greater(x, tf.constant(15.0))
    smallerEqualFiveteen = tf.less_equal(x, tf.constant(15.0))
    x = tf.where(smallerEqualZero, 1.359140915 *
     tf.math.exp(tf.where(smallerEqualZero, (x-1), 0)),
     tf.where(greaterFiveteen, 1 - 1/(109.0858178 * x - 1403.359435),
     0.03 * tf.math.log(tf.where(greaterZero,
     tf.where(smallerEqualFiveteen, (1000000 * x + 1), 0), 0)) + 0.5))

    return x
```

Nun können wir die eigene Aktivierungsfunktion auf das Modell übertragen und schauen, ob sich die Trainingsergebnisse verbessern.

```
model = Sequential([
    Dense(2, activation=custom_activation),
    Dense(1, activation=custom_activation),
])
```

Auch wenn wir so ein besseres Loss bekommen, ist noch nicht von einem Erfolg zu sprechen. Die Vorhersagen des Modells sind immer noch weit weg von wirklicher Division.

2.4. Hyper-Parameter

In der Hoffnung, bessere Ergebnisse zu erzielen, können wir die Hyper-Parameter optimieren. Dies tun wir über Talos. Wir können verschiedene Werte für die Parameter ausprobieren und am Ende das Zusammenspiel mit dem niedrigsten MAE wählen. Zusätzlich erstellen wir in unserem Modell eine Dropout-Schicht.

```
p = {
    'batch_size': [1, 4, 16, 32, 64],
    'dropout': [0, 0.0001, 0.0002, 0.0005],
    'epochs': [90],
    'optimizers': ['rmsprop', 'ftrl', 'nadam',
     'adam', 'sgd', 'adadelta', 'adamax'],
}

model = Sequential([
        Dense(2, activation=custom_activation),
        Dropout(params['dropout']),
        Dense(1, activation=custom_activation),
    ])

talos.Scan(X_train, Y_train, model=division_model,
params=p, experiment_name='division')
```

Als optimales Ergebnis bekommen wir für den Optimizer Nadam, als Batchsize 4 und ein hohes Dropout von 0.0005. Gerade durch die Wahl des neuen Optimizers lässt sich das Loss weiter drücken. Trotzdem sind wir von dem Taschenrechner-ähnlichen Ergebnis der theoretischen Python-Funktion noch weit entfernt.

2.5. Eigene Differenzierung

Aus irgendeinem Grund scheint die Backpropagation unseres Modells also noch nicht perfekt zu funktionieren. Lasst uns daher probieren, die Differenzierung unserer Aktivierungsfunktion eigenhändig zu berechnen. Dies können wir tun mit dem tensorflow decorator @tf.custom_gradient.

```
@tf.custom_gradient
def custom_activation(x):
    smallerEqualZero = tf.less_equal(x, tf.constant(0.0))
    greaterZero = tf.greater(x, tf.constant(0.0))
    greaterFiveteen = tf.greater(x, tf.constant(15.0))
    smallerEqualFiveteen = tf.less_equal(x, tf.constant(15.0))
    result = tf.where(smallerEqualZero, 1.359140915 *
    tf.math.exp(tf.where(smallerEqualZero, (x-1), 0)),
    tf.where(greaterFiveteen, 1 - 1/(109.0858178 * x - 1403.359435),
    0.03 * tf.math.log(tf.where(greaterZero,
    tf.where(smallerEqualFiveteen, (1000000 * x + 1), 0), 0)) + 0.5))

    def grad(dy):
        return dy * tf.where(smallerEqualZero, 1.359140915 *
         tf.math.exp(tf.where(smallerEqualZero, (x-1), 0)),
        tf.where(greaterFiveteen, 501379254(4596191*(501379254
        * x / 4596191 - 280671887 / 200000)**2), 30000/(1000000*x+1)))

    return result, grad
```

Aber wie erwartet, ändert sich das Loss dadurch nicht.

2.6. Einspielen eigener Gewichte

Wenn es auf diesen Wegen nicht klappt, können wir die Gewichte aus dem Vortrag auf die Schichten unseres neuronalen Netzes übertragen.

```
layer0 = [np.array([[0.000001, 0.0], [0.0, 0.000001]],
dtype=np.float32), np.array([-0.000001, -0.000001], dtype=np.float32)]
layer2 = [np.array([[33.3333], [-33.3333]],
dtype=np.float32), np.array([-3.912023], dtype=np.float32)]

model.layers[0].set_weights(layer0)
model.layers[2].set_weights(layer2)
```

Nun bekommen wir ein MAE, das der theoretischen Python-Funktion in nichts mehr nachsteht.

3. Streamlit

Um unsere Arbeit zu präsentieren, habe ich eine Web-App mit Streamlit erstellt, die es jedem erlaubt, seine eigenen Divisionsaufgaben durch mehrere geladene Modelle laufen zu lassen.

Keras Custom Grad

```
@tf.custom_gradient
def custom_activation(x):
    smallerEqualZero = tf.less_equal(x, tf.constant(0.0))
    greaterZero = tf.greater(x, tf.constant(0.0))
    greaterFiveteen = tf.greater(x, tf.constant(15.0))
    smallerEqualFiveteen = tf.less_equal(x, tf.constant(15.0))
    result = tf.where(smallerEqualZero, 1.359140915 * tf.math.exp(tf.where(sma
            tf.where(greaterFiveteen, 1 - 1/(109.0858178 * x - 1403.359435),
            0.03 * tf.math.log(tf.where(greaterZero, tf.where(smallerEqualFive

    def grad(dy):
        return dy * tf.where(smallerEqualZero, 1.359140915 * tf.math.exp(tf.wh
            tf.where(greaterFiveteen, 501379254/(4596191*(501379254 * x / 459
            30000/(1000000*x+1)))

    return result, grad
```

Probiert es unter https://share.streamlit.io/foersterrobert/division/DivisionStreamlit.py gerne selbst aus.

Der vollständige Code ist über https://github.com/foersterrobert/division zu finden.