Data Mining & Machine Learning

Computer Exercise 6 - Neural Networks & PyTorch

Steinarr Hrafn Höskuldsson

September 2022

Section 1

I moved the training to my laptops GPU and added significantly more neurons to both the convolutional layers and the first linear layer. The total accuracy of the network increased from 52% as it was in the training example to 65%. The misclassification rates for each for each category was plotted and can be seen in Figure ??

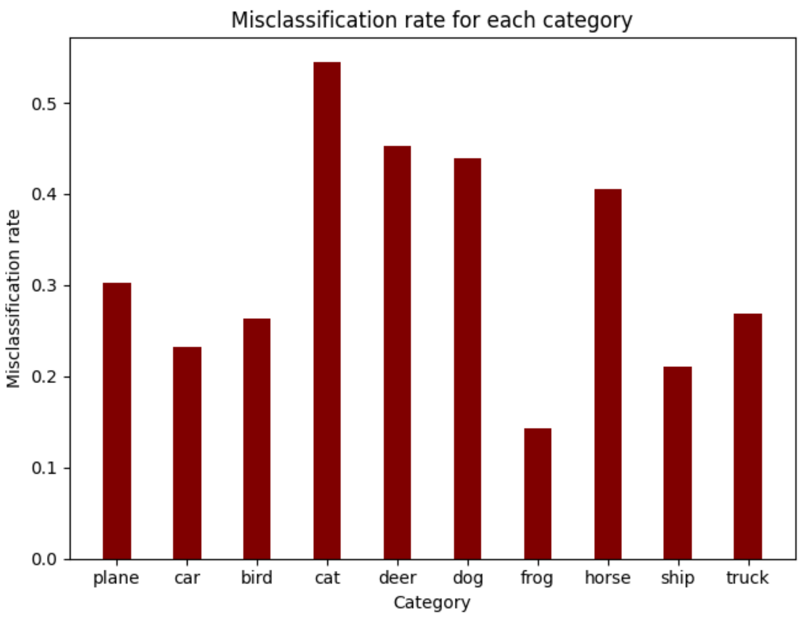

Figure 1: Misclassification rate of each category

The Confusion matrix was calculated.

$$
\mathrm{CM} = \begin{bmatrix}
 & plane & car & bird & cat & deer & dog & frog & horse & ship & truck \\
plane & 765 & 23 & 16 & 29 & 20 & 3 & 4 & 21 & 74 & 45 \\
car & 16 & 839 & 3 & 13 & 4 & 4 & 3 & 4 & 22 & 92 \\
bird & 113 & 20 & 350 & 98 & 147 & 110 & 43 & 81 & 19 & 19 \\
cat & 33 & 13 & 25 & 499 & 75 & 205 & 21 & 87 & 19 & 23 \\
deer & 15 & 4 & 34 & 70 & 595 & 37 & 16 & 206 & 19 & 4 \\
dog & 13 & 6 & 17 & 169 & 49 & 607 & 5 & 116 & 5 & 13 \\
frog & 7 & 14 & 21 & 156 & 148 & 37 & 578 & 18 & 15 & 6 \\
horse & 17 & 0 & 6 & 26 & 41 & 63 & 2 & 817 & 3 & 25 \\
ship & 80 & 43 & 2 & 22 & 5 & 10 & 0 & 3 & 787 & 48 \\
truck & 38 & 131 & 1 & 14 & 4 & 6 & 2 & 22 & 34 & 748
\end{bmatrix}
$$

In the confusion matrix one can spot that the most common misclassifcations were mix ups of dog/cat, bird/plane, car/truck, horse/deer/frog which suggests the network might be relying on the background to some extent to classify. Perhaps if the training data also included images with the background extracted the networks could performance could be improved.

The training of the network in PyTorch relied heavily on the AutoGrad which is PyTorch's automatic differentation engine. It keeps track of operations performed on the dataflow and knows the derivative of those events. When needed, it can apply the chain rule to compute the partial derivative in question.

Section 2

RNN stand for Recursive Neural Network. The recursiveness comes from the fact that the part of the input into the network is the output from the last run. This allows the network to remember what was going on. RNN's are very useful in situations where something is happening over time and the network has to remember what has already happened. A good example of such a situation is working with text. A very important part of having a conversation or completing a sentence is remembering what has previously been said. Other use cases for RNNs are for example video analysis, speech recognition and sequence prediction.

A RNN network was trained based on the example from Pytorch's tutorial. In my opinion the RNN model in Pytorch's example did not perform very well. Although the made up names are usually phonetically viable, they usually did not sound like real names.

Independent Section

All icelandic given names were downloaded from opinskra.is. A short python program was written to parse the valid ones into a .txt file. The Name generation example was then run again and the output generated for Icelandic names was as follows:

- Sari
- Tring
- Ering
- Irin
- Nari
- Arrin
- Rongani
- Rimar

And to a native Icelandic speaker such as myself, none of those sound like an actual name except for the last one, Rimar, which fooled me, I had to look up that it is indeed not a name.

Appendix

A Code

```
# author: Steinarr Hrafn

from typing import Union
import numpy as np
from matplotlib import pyplot as plt
from tools import load_iris, split_train_test


def sigmoid(x: float) -> float:
    '''
    Calculate the sigmoid of x
    '''
    if isinstance(x, np.ndarray):
        x[x<-100] = -100
    elif x < -100:
        return 0
    return 1/(1+np.exp(-x))


def d_sigmoid(x: float) -> float:
    '''
    Calculate the derivative of the sigmoid of x.
    '''
    return sigmoid(x)*(1-sigmoid(x))


def perceptron(
    x: np.ndarray,
    w: np.ndarray
) -> Union[float, float]:
    '''
    Return the weighted sum of x and w as well as
    the result of applying the sigmoid activation
    to the weighted sum
    '''
    return np.sum(w*x), sigmoid(np.sum(w*x))


def ffnn(
    x: np.ndarray,
    M: int,
    K: int,
    W1: np.ndarray,
    W2: np.ndarray,
) -> Union[np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    '''
    Computes the output and hidden layer variables for a
    single hidden layer feed-forward neural network.
    '''
    z0 = np.hstack(([1], x))
    a1 = np.sum(z0*np.transpose(W1), axis=1)
    z1 = np.hstack(([1], sigmoid(a1)))
    a2 = np.sum(z1*np.transpose(W2), axis=1)
    y = sigmoid(a2)
    # a2 = np.sum(z1*np.transpose(W2[:-1, :])) + W2[-1, :]
```

```python

    return y, z0, z1, a1, a2


def backprop(
    x: np.ndarray,
    target_y: np.ndarray,
    M: int,
    K: int,
    W1: np.ndarray,
    W2: np.ndarray
) -> Union[np.ndarray, np.ndarray, np.ndarray]:
    '''
    Perform the backpropagation on given weights W1 and W2
    for the given input pair x, target_y
    '''
    y, z0, z1, a1, a2 = ffnn(x, M, K, W1, W2)
    dk   = y - target_y
    # print( a2.shape, dk.shape, W2.shape, )
    # print(W2)
    dj = d_sigmoid(a1)*np.sum(dk*W2[1:, :], axis=1)

    dE1 = dj*z0[..., None]
    dE2 = dk*z1[..., None]
    return y, dE1, dE2


def cross_entropy(ts, ys):
    # assume one hot encoding
    return -np.sum(ts*np.log(ys) + (1-ts)*np.log(1-ys))

def one_hot(t, c=3):
    ret = np.zeros((c))
    ret[t] = 1
    return ret

def hot_one(one):
    return np.argmax(one, axis=-1)

def compare_one_hots(y1, y2):
    return np.argmax(y1) == np.argmax(y2)

def train_nn(
    X_train: np.ndarray,
    t_train: np.ndarray,
    M: int,
    K: int,
    W1: np.ndarray,
    W2: np.ndarray,
    iterations: int,
    eta: float
) -> Union[np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    '''
    Train a network by:
    1. forward propagating an input feature through the network
    2. Calculate the error between the prediction the network
    made and the actual target
    3. Backpropagating the error through the network to adjust
    the weights.
    '''
```

```python
    # print(t_train)
    W1tr = W1.copy()
    W2tr = W2.copy()
    N = X_train.shape[0]
    misclassification_rate = []
    E_total = []
    guesses = []
    # loop through iterations
    for iteration in range(iterations):
        E = 0
        misclassifications = 0
        dE1_total = np.zeros(W1tr.shape)
        dE2_total = np.zeros(W2tr.shape)
        # loop through training set
        for x, y_target in zip(X_train, t_train):
            y_target = one_hot(y_target)
            y, dE1, dE2 = backprop(x, y_target, 0, 0, W1tr, W2tr)
            # y = hot_one(y)
            dE1_total += dE1
            dE2_total += dE2
            if iteration == iterations -1:
                guesses.append(hot_one(y))
            # if iteration == 0 or iteration == iterations-1:
                # print(y_target, y, compare_one_hots(y_target, y))
            E += cross_entropy(y_target, y)
            misclassifications += not compare_one_hots(y_target, y)
        W1tr -= eta*dE1_total/N
        W2tr -= eta*dE2_total/N
        # guesses = np.array(guesses)
        E_total.append(E/N)
        misclassification_rate.append(misclassifications/N)
    return W1tr, W2tr, E_total, misclassification_rate, guesses


def test_nn(
    X: np.ndarray,
    M: int,
    K: int,
    W1: np.ndarray,
    W2: np.ndarray
) -> np.ndarray:
    '''
    Return the predictions made by a network for all features
    in the test set X.
    '''
    return np.array([hot_one(ffnn(x, 0, 0, W1, W2)[0]) for x in X])


if __name__ == '__main__':

    print(f"\n\n{'-' * 20}\n\t Section 1.1 \n")
    print(f"{sigmoid(0.5)=}")
    print(f"{d_sigmoid(0.2)=}")

    print(f"\n\n{'-' * 20}\n\t Section 1.2\n")
    print(f"{perceptron(np.array([1.0, 2.3, 1.9]),np.array([0.2,0.3,0.1]))=}")
    print(f"{perceptron(np.array([0.2,0.4]),np.array([0.1,0.4]))=}")

    np.random.seed(34545)
    print(f"\n\n{'-' * 20}\n\t Section 1.3\n")
```

```python
    features, targets, classes = load_iris()
    (train_features, train_targets), (test_features, test_targets) = \
        split_train_test(features, targets)
    # initialize the random generator to get repeatable results
    np.random.seed(1234)

    # Take one point:
    x = train_features[0, :]
    K = 3  # number of classes
    M = 10
    D = 4
    # Initialize two random weight matrices
    W1 = 2 * np.random.rand(D + 1, M) - 1
    W2 = 2 * np.random.rand(M + 1, K) - 1
    y, z0, z1, a1, a2 = ffnn(x, M, K, W1, W2)

    print(f"{y=}\n{z0=}\n{z1=}\n{a1=}\n{a2=}")


    print(f"\n\n{'-' * 20}\n\t Section 1.4\n")
    # initialize random generator to get predictable results
    np.random.seed(42)

    K = 3  # number of classes
    M = 6
    D = train_features.shape[1]

    x = features[0, :]

    # create one-hot target for the feature
    target_y = np.zeros(K)
    target_y[targets[0]] = 1.0

    # Initialize two random weight matrices
    W1 = 2 * np.random.rand(D + 1, M) - 1
    W2 = 2 * np.random.rand(M + 1, K) - 1

    y, dE1, dE2 = backprop(x, target_y, M, K, W1, W2)

    print(f"{y=}\n{dE1=}\n{dE2=}")


    print(f"\n\n{'-' * 20}\n\t Section 2.1\n")
    # initialize the random seed to get predictable results
    np.random.seed(1234)

    K = 3  # number of classes
    M = 6
    D = train_features.shape[1]

    # Initialize two random weight matrices
    W1 = 2 * np.random.rand(D + 1, M) - 1
    W2 = 2 * np.random.rand(M + 1, K) - 1
    W1tr, W2tr, Etotal, misclassification_rate, last_guesses = train_nn(
        train_features[:20, :], train_targets[:20], M, K, W1, W2, 500, 0.1)

    print(f"W1tr = \n{W1tr}\n")
    print(f"W2tr = \n{W2tr}\n")
    print(f"Etotal = \n{Etotal[:10]}\n...\n{Etotal[-10:]}\n")
    print(
```

```
        f"misclassification_rate = \n{misclassification_rate[:10]}\n...\n{
    misclassification_rate[-10:]}\n")

    print(f"last_guesses = \n{last_guesses[:10]}\n...\n{last_guesses[-10:]}\n")
    print(train_targets[:20])

    print(f"\n\n{'-' * 20}\n\t Section 2.2\n")
    W1tr, W2tr, Etotal, misclassification_rate, last_guesses = train_nn(
        train_features, train_targets, M, K, W1, W2, 500, 0.1)
    guesses = test_nn(test_features, 0, 0, W1tr, W2tr)
    print(f"{guesses=}")

    print(f"\n\n{'-' * 20}\n\t Section 2.3\n")
    accuracy = np.count_nonzero(test_targets==guesses)/len(guesses)

    matrix = np.zeros((3, 3), int)
    for i, a in enumerate(classes):
        for j, p in enumerate(classes):
            matrix[j, i] = np.count_nonzero(guesses[np.where(test_targets == a)] == p)
    print(f"{accuracy=:.1%}")
    print(f"Confusion matrix = \n{matrix}")

    def format_matrix(matrix, environment="bmatrix", formatter=str):
        """Format a matrix using LaTeX syntax"""

        if not isinstance(matrix, np.ndarray):
            try:
                matrix = np.array(matrix)
            except Exception:
                raise TypeError("Could not convert to Numpy array")

        if len(shape := matrix.shape) == 1:
            matrix = matrix.reshape(1, shape[0])
        elif len(shape) > 2:
            raise ValueError("Array must be 2 dimensional")

        body_lines = [" & ".join(map(formatter, row)) for row in matrix]

        body = "\\\\\n".join(body_lines)
        return f"""\\begin{{{environment}}}
    {body}
    \\end{{{environment}}}"""
    print(format_matrix(matrix))


    plt.plot(Etotal, label="E_total")
    plt.plot(misclassification_rate, label="misclassification_rate")
    plt.xlabel("Iterations")
    plt.legend()
    plt.show()


    print(f"\n\n{'-' * 20}\n\t Independent Section\n")

    # run50 = []
    # for _ in range(50):
    #
    #     (train_features, train_targets), (test_features, test_targets) = \
    #         split_train_test(features, targets)
    #
```

```
#
#     def test_performance(eta, iterations, W1, W2):
#         W1tr, W2tr, Etotal, misclassification_rate, last_guesses = train_nn(
train_features[:20, :], train_targets[:20], M, K, W1, W2, iterations, eta)
#         guesses = test_nn(test_features, 0, 0, W1tr, W2tr)
#         return np.count_nonzero(test_targets == guesses) / len(guesses)
#
etas = np.arange(0.02, 2.02, 0.05)
iterations = np.array(range(50, 1000, 50))
#     results = []
#     for eta in etas:
#         print(eta)
#         r =[]
#         for n in iterations:
#             r.append(test_performance(eta, n, W1, W2))
#         results.append(r)
#     results = np.array(results)
#     run50.append(results)
# with open("indep_data_50.npy", 'wb') as f:
#     np.save(f, np.array(run50))
with open("05_backprop/indep_data_50.npy", 'rb') as f:
    results = np.load(f)
results = (np.average(results, axis=0) - 2*np.std(results, axis=0))

# results = np.std(results, axis=0)
print(results)
print(etas[results.argmax()//results.shape[1]],
      iterations[results.argmax() % results.shape[1]])

fig, ax = plt.subplots()


c = ax.pcolormesh(iterations, etas, results, cmap='RdBu', vmin=0.5, vmax=1, shading="
nearest")

ax.set_title('Heatmap of accuracy, changing eta and number of iterations \n avg - 2*std
over 50 different train/test splits')

ax.set_xlabel("Number of Iterations")
ax.set_ylabel("Learning rate [eta]")

# set the limits of the plot to the limits of the data
# ax.axis([x.min(), x.max(), y.min(), y.max()])
fig.colorbar(c, ax=ax)

plt.show()
```