ECE421

Assignment 2

Marilyn D'Souza - 1002368831

Due: 6 March 2020

1 Neural Networks Using Numpy

1.1 Helper Functions

1.1.1 Code

```
def relu(x):
    return max(x, 0)

def softmax(x):
    x = x - np.max(x)
    return np.divide(np.exp(x), np.sum(np.exp(x)))

def computeLayer(X, W, b):
    return np.matmul(np.transpose(W), X) + b

def CE(target, prediction):
    return -1*np.mean(np.multiply(target, np.log(prediction)))

def gradCE(target, prediction):
    return  target - prediction
```

1.1.2 Deriving the Gradient of CE

$$L = -1(y_1 log(p_1) + ... + y_k log(p_k))$$

$$L = -1\left(y_1 log\left(\frac{e^{o_1}}{e^{o_1} + ... + e^{o_k}}\right) + ... + y_k log\left(\frac{e^{o_k}}{e^{o_1} + ... + e^{o_k}}\right)\right)$$

$$L = y_1(log(e^{o_1} + ... + e^{o_k}) - o_1) + ... + y_k(log(e^{o_1} + ... + e^{o_k}) - o_k)$$

Let's take the partial derivative with respect to o_1.

$$\frac{\partial L}{\partial o_1} = y_1\left(\frac{e^{o_1}}{e^{o_1} + ... + e^{o_k}} - 1\right) + y_2\left(\frac{e^{o_1}}{e^{o_1} + ... + e^{o_k}}\right) + ... + y_k\left(\frac{e^{o_1}}{e^{o_1} + ... + e^{o_k}}\right)$$

$$\frac{\partial L}{\partial o_1} = y_1(p_1 - 1) + y_2 p_1 + ... + y_k p_1$$
$$\frac{\partial L}{\partial o_1} = y_1 p_1 + y_2 p_1 + ... + y_k p_1 - y_1$$
One of $y_1, ..., y_k$ is labelled 1. The rest are labelled 0. Then,

$$\frac{\partial L}{\partial o_1} = p_1 - y_1$$

In general,

$$\delta^o = \frac{\partial L}{\partial o} = p - y$$

1.2 Backpropagation Derivation

1. The gradient of the loss with respect to the output layer weights.

$$\frac{\partial L}{\partial W_o} = \frac{\partial o}{\partial W_o}\left(\frac{\partial L}{\partial o}\right)^T$$
$$\frac{\partial L}{\partial W_o} = \frac{\partial (W_o^T h + b_o)}{\partial W_o}(p - y)^T$$
$$\frac{\partial L}{\partial W_o} = h(p - y)^T$$

2. The gradient of the loss with respect to the output layer biases.

$$\frac{\partial L}{\partial b_o} = \frac{\partial o}{\partial b_o}\left(\frac{\partial L}{\partial o}\right)^T$$
$$\frac{\partial L}{\partial b_o} = \frac{\partial (W_o^T h + b_o)}{\partial b_o}(p - y)^T$$
$$\frac{\partial L}{\partial b_o} = (p - y)^T$$

3. The gradient of the loss with respect to the hidden layer weights.

$$\frac{\partial L}{\partial W_h} = \frac{\partial (W_h^T x + b_h)}{\partial W_h}\left(\frac{\partial L}{\partial (W_h^T + b_h)}\right)^T$$
$$\frac{\partial L}{\partial W_h} = x(\delta^h)^T$$
$$\delta^h = \theta'(W_h^T x + b_h) \bigotimes W^o \delta^o$$
$$\theta(s) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \qquad \theta'(s) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases}$$
$$\frac{\partial L}{\partial W_h} = x(\delta^h)^T = x(\theta'(W_h^T x + b_h) \bigotimes W^o(p - y))^T$$

4. The gradient of the loss with respect to the hidden layer biases.

$$\frac{\partial L}{\partial b_h} = \frac{\partial (W_h^T x + b_h)}{\partial b_h} \left(\frac{\partial L}{\partial (W_h^T + b_h)} \right)^T$$

$$\frac{\partial L}{\partial b_h} = (\delta^h)^T$$

$$\frac{\partial L}{\partial W_h} = (\delta^h)^T = (\theta'(W_h^T x + b_h) \bigotimes W^o(p-y))^T$$

1.3 Learning

The parameters were initialized as follows:

```
H = 1000
gamma = 0.99
alpha = 10**-6
epochs = 200
```

Xavier initialization was used for the weights and biases:

```
    Wh = np.random.normal(0, 2/(784 + H), (784, H))
    bh = np.random.normal(0, 2/(784 + H), (H, 1))
    Wo = np.random.normal(0, 2/(H + 10), (H, 10))
    bo = np.random.normal(0, 2/(784 + H), (10, 1))
```

The following code was written to find roughly optimal weights and biases, as well as their loss and accuracy scores, given values for H (the size of the hidden layer), gamma (momentum), alpha (learning rate), the number of epochs, and a set of input data X with labels Y.

```
def optimize(H, gamma, alpha, epochs, X, Y):

    Wh = np.random.normal(0, 2/(784 + H), (784, H))
    bh = np.random.normal(0, 2/(784 + H), (H, 1))
    Wo = np.random.normal(0, 2/(H + 10), (H, 10))
    bo = np.random.normal(0, 2/(784 + H), (10, 1))

    old_vWh = vWh = old_vbh = vbh = old_vWo = vWo = old_vbo = vbo = 10**-5

    loss = []
    accuracy = []

    for i in range(epochs):

        h = relu(computeLayer(np.transpose(X), Wh, bh))
        o = computeLayer(h, Wo, bo)
```

```
        p = np.apply_along_axis(softmax, 0, o)

        loss.append(CE(np.transpose(Y), p))
        accuracy.append(np.sum(np.argmax(p, 0) == np.argmax(np.transpose(Y),
        ↪  0))/len(X))

        gWo = np.matmul(h, np.transpose(p - np.transpose(Y)))
        gbo = np.sum(np.transpose(p - np.transpose(Y)), 0, keepdims = True)
        gWh = np.matmul(np.transpose(X),
        ↪  np.transpose((np.multiply(np.heaviside
        ↪  (computeLayer(np.transpose(X), Wh, bh), 0), np.matmul(Wo, p -
        ↪  np.transpose(Y))))))
        gbh = np.sum(np.transpose((np.multiply(np.heaviside
        ↪  (computeLayer(np.transpose(X), Wh, bh), 0), np.matmul(Wo, p -
        ↪  np.transpose(Y))))), 0, keepdims = True)

        vWh = gamma*old_vWh + alpha*gWh
        vbh = gamma*old_vbh + alpha*gbh
        vWo = gamma*old_vWo + alpha*gWo
        vbo = gamma*old_vbo + alpha*gbo

        old_vWh, old_vbh, old_vWo, old_vbo = vWh, vbh, vWo, vbo

        Wh = Wh - vWh
        bh = bh - np.transpose(vbh)
        Wo = Wo - vWo
        bo = bo - np.transpose(vbo)

    plt.plot(range(len(loss)), loss)
    plt.xlabel("EPOCH")
    plt.ylabel("LOSS")

    plt.plot(range(len(accuracy)), accuracy)
    plt.xlabel("EPOCH")
    plt.ylabel("ACCURACY")

    plt.legend(["Loss", "Accuracy"])

    return Wh, bh, Wo, bo, loss, accuracy
```

We use this function to calculate loss and accuracy for the training set:

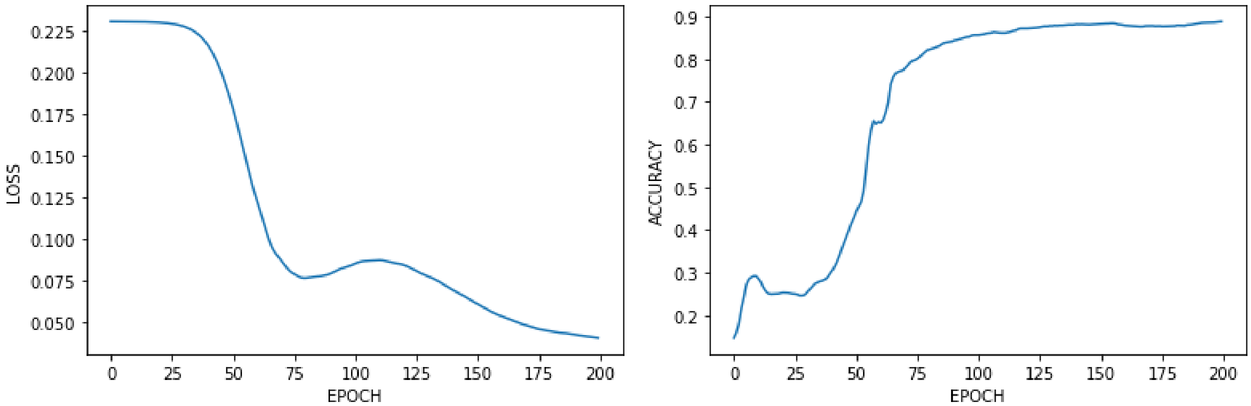

Figure 1: *Training Loss and Accuracy for* $\alpha = 10^{-}5$, $\gamma = 0.99$

Final loss: 0.040236805632313334.
Final accuracy: 0.8908.

Loss and accuracy for the validation set:

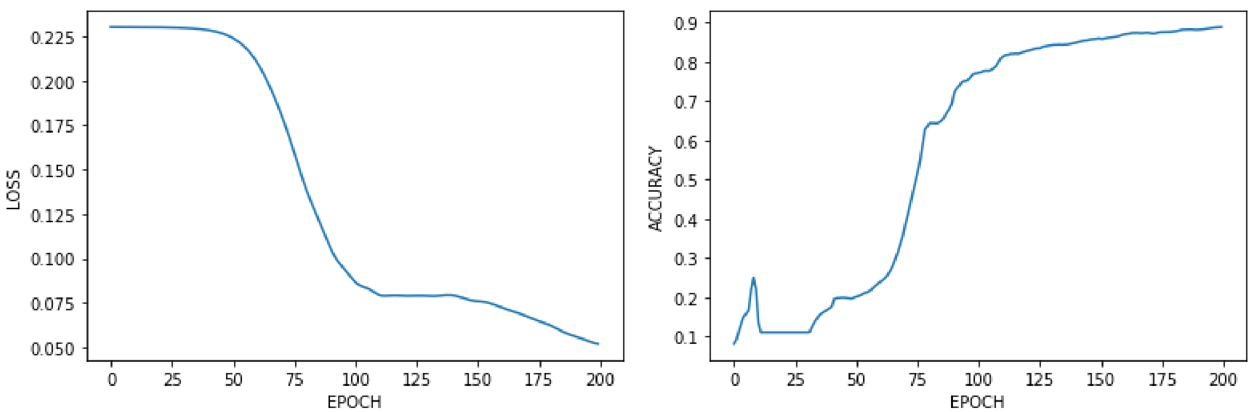

Figure 2: *Validation Loss and Accuracy for* $\alpha = 10^{-}5$, $\gamma = 0.99$

Final loss: 0.051857979270676285.
Final accuracy: 0.8875.

Loss and accuracy for the test set:

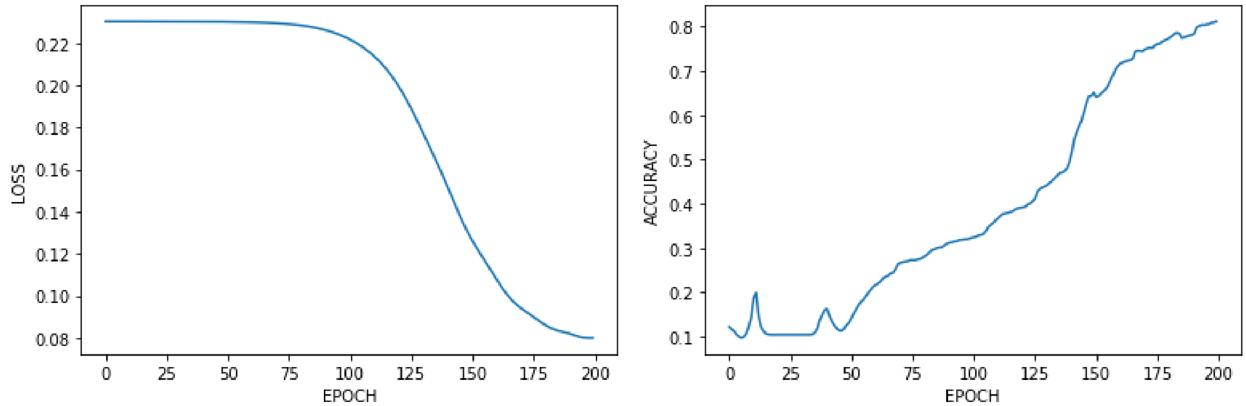

Figure 3: *Validation Loss and Accuracy for* $\alpha = 10^{-}5$, $\gamma = 0.99$

Final loss: 0.07818691179048143.
Final accuracy: 0.8043318649045521.

1.4 Hyperparameter Investigation

Loss and accuracy for the training set, H = 100:

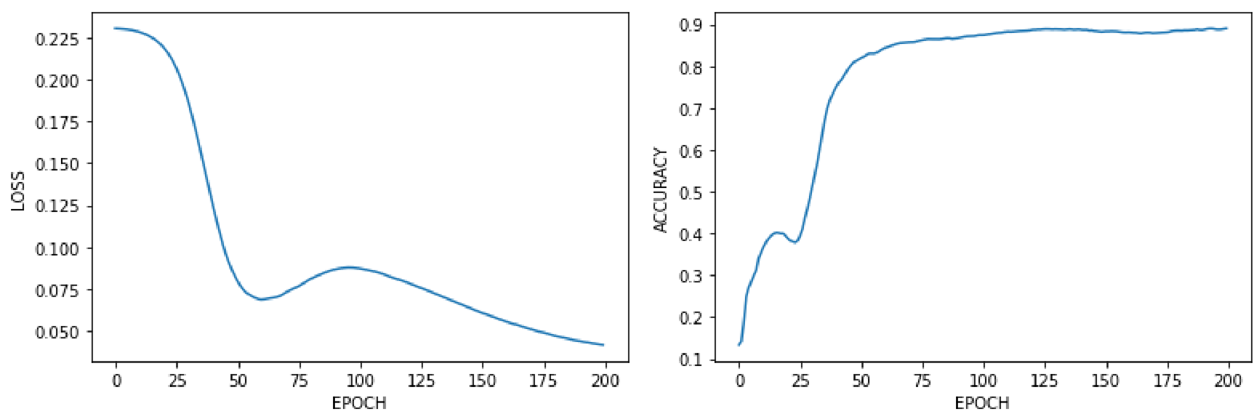

Figure 4: *Training Loss and Accuracy for* $\alpha = 10^{-}5$, $\gamma = 0.99$, *H = 100*

Final loss: 0.041785957761196306.
Final accuracy: 0.8901.

Loss and accuracy for the validation set, H = 100:

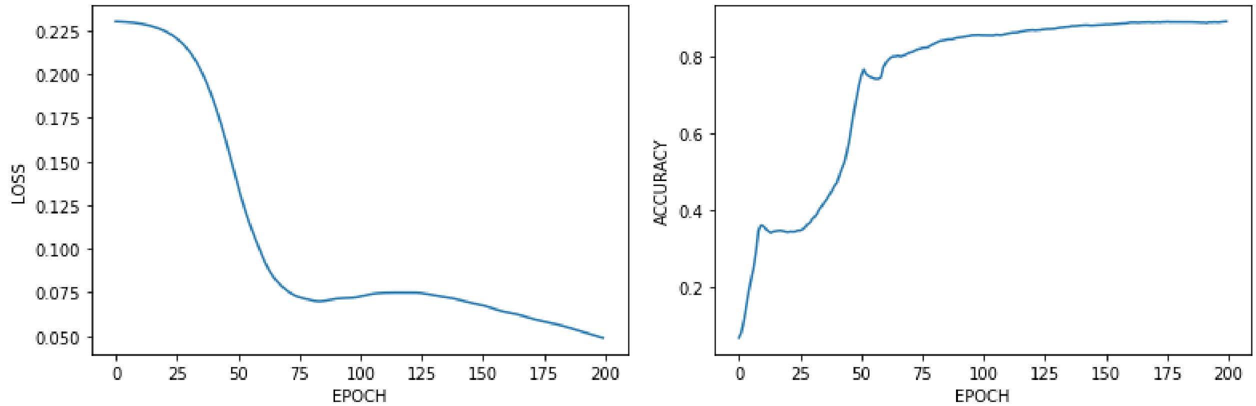

Figure 5: *Validation Loss and Accuracy for* $\alpha = 10^{-}5$, $\gamma = 0.99$, H = *100*

Final loss: 0.048854359933197464.
Final accuracy: 0.8928333333333334.

Loss and accuracy for the test set, H = 100:

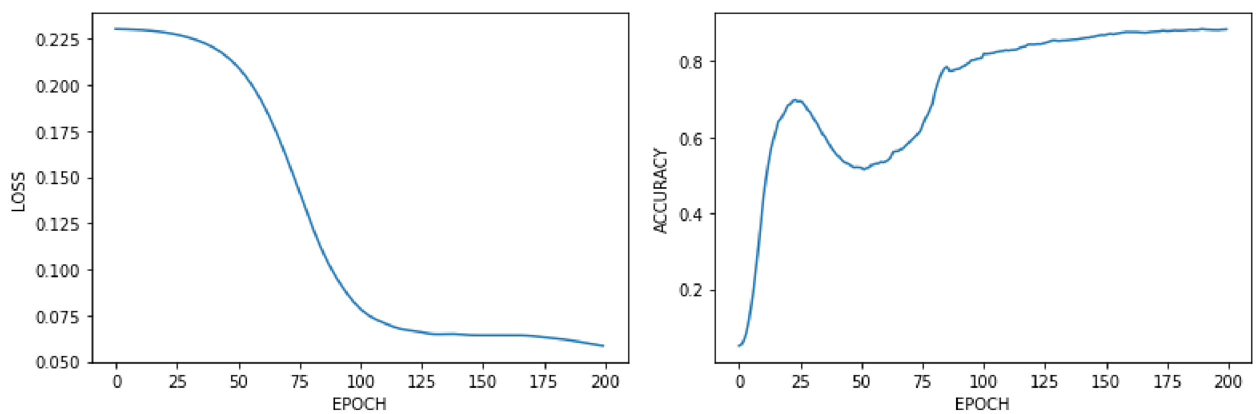

Figure 6: *Validation Loss and Accuracy for* $\alpha = 10^{-}5$, $\gamma = 0.99$, H = *100*

Final loss: 0.05841501603696067.
Final accuracy: 0.8836270190895742.

Loss and accuracy for the training set, H = 500:

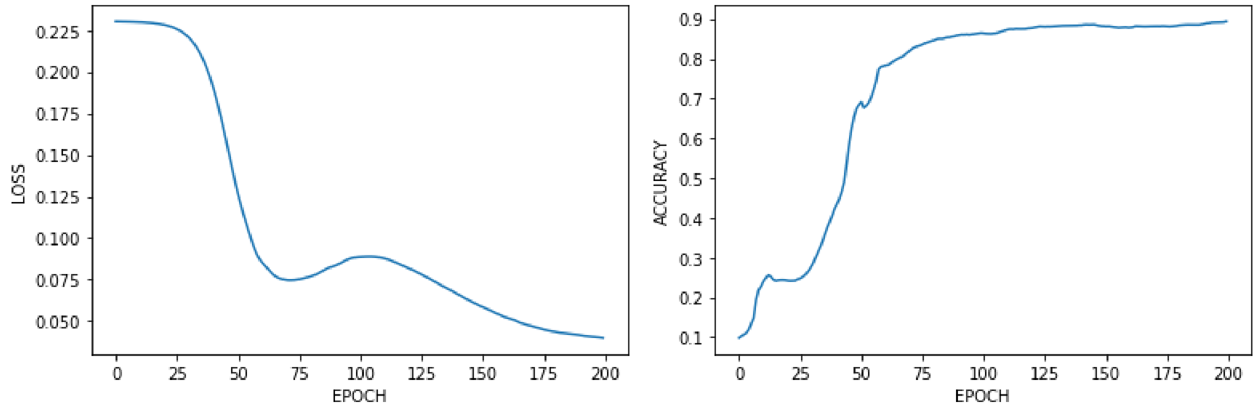

Figure 7: *Training Loss and Accuracy for* $\alpha = 10^{-}5$, $\gamma = 0.99$, *H = 500*

Final loss: 0.039821532798109814.
Final accuracy: 0.8936.

Loss and accuracy for the validation set, H = 500:

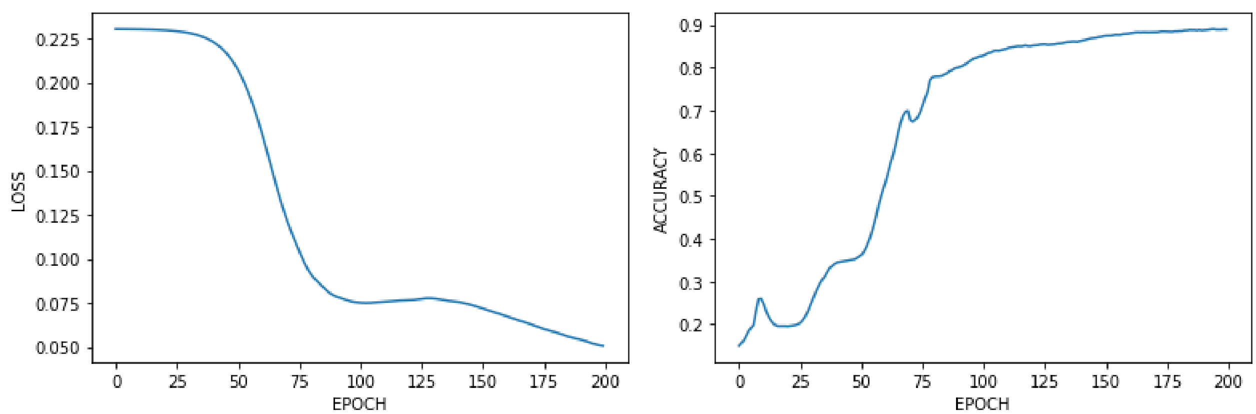

Figure 8: *Validation Loss and Accuracy for* $\alpha = 10^{-}5$, $\gamma = 0.99$, *H = 500*

Final loss: 0.050655216981619 72.
Final accuracy: 0.8893333333333333.

Loss and accuracy for the test set, H = 500:

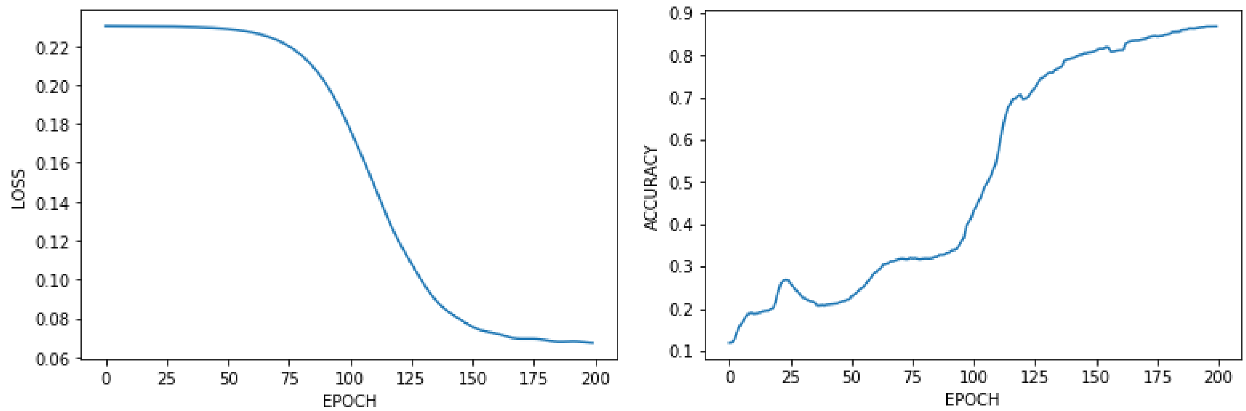

Figure 9: *Test Loss and Accuracy for* $\alpha = 10^{-}5$, $\gamma = 0.99$, *H = 500*

Final loss: 0.06730111608315559.
Final accuracy: 0.868575624082232.

Loss and accuracy for the training set, H = 2000:

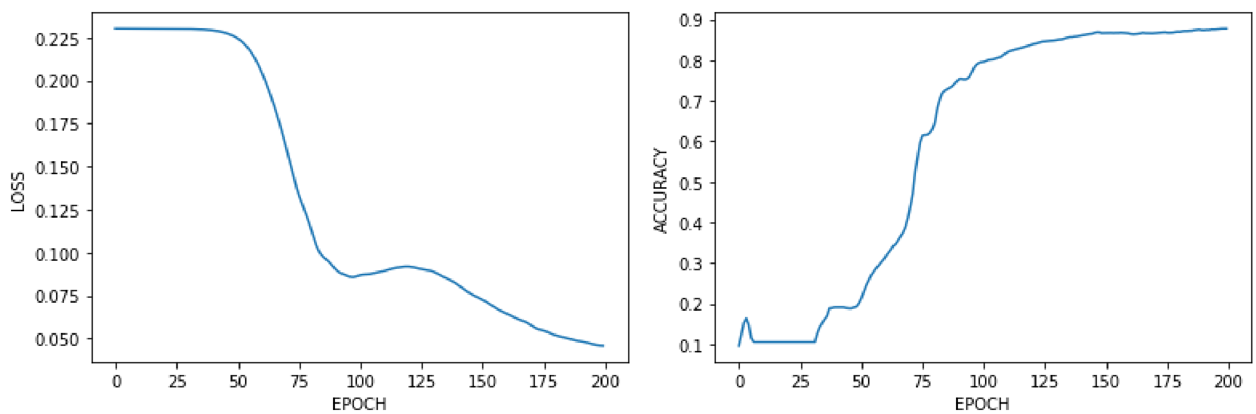

Figure 10: *Test Loss and Accuracy for* $\alpha = 10^{-}5$, $\gamma = 0.99$, *H = 2000*

Final loss: 0.06730111608315559.
Final accuracy: 0.868575624082232.

Loss and accuracy for the validation set, H = 2000:

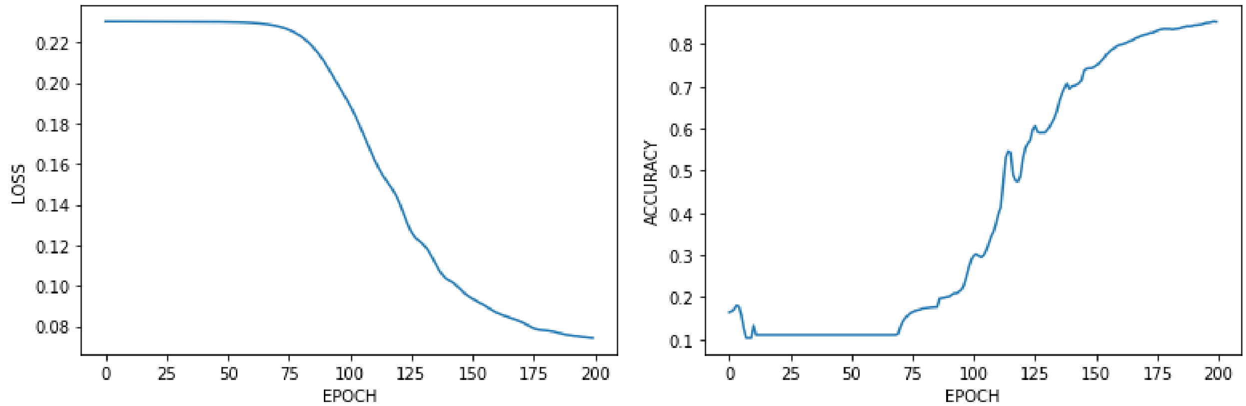

Figure 11: *Validation Loss and Accuracy for* $\alpha = 10^{-}5$, $\gamma = 0.99$, *H = 2000*

Final loss: 0.07426464633510227.
Final accuracy: 0.8531666666666666.

Loss and accuracy for the test set, H = 2000:

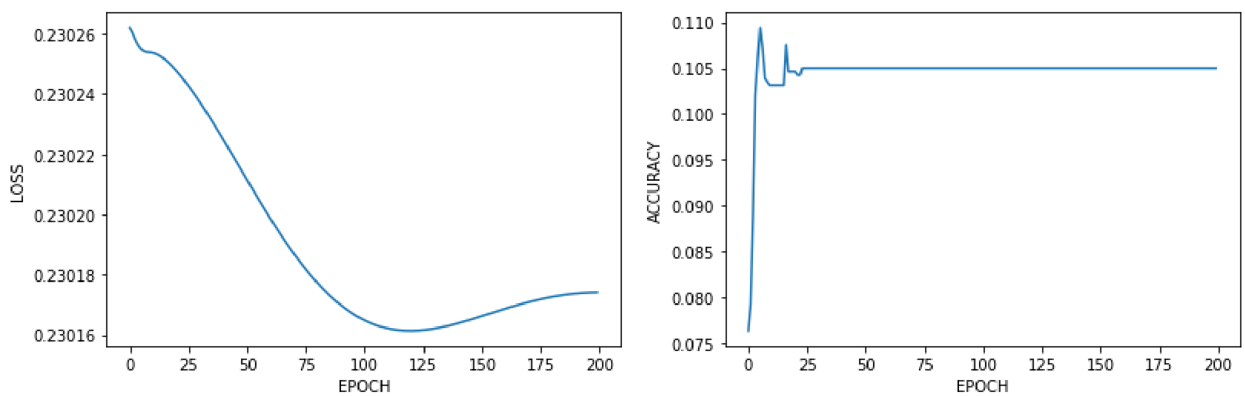

Figure 12: *Test Loss and Accuracy for* $\alpha = 10^{-}5$, $\gamma = 0.99$, *H = 2000*

Final loss: 0.2301740905705153.
Final accuracy: 0.10499265785609398.

H	Training	Validation	Test
100	0.041785	0.048854	0.058415
500	0.039821	0.050655	0.067301
1000	0.040236	0.051857	0.078186
2000	0.067301	0.074264	0.23017

Table 1: *CE is minimal at H = 100 and H = 500 and increases for H = 1000 and H = 2000, presumably because of overfitting.*

H	Training	Validation	Test
100	0.041785957761196306	0.048854359933197464	0.05841501603696067
500	0.039821532798109814	0.05065521698161972	0.06730111608315559
1000	0.040236805632313334	0.051857979270676285	0.07818691179048143
2000	0.06730111608315559	0.07426464633510227	0.2301740905705153

Table 2: *Accuracy is minimal at H = 100 and H = 500 and increases for H = 1000 and H = 2000, presumably because of overfitting.*

2 Neural Networks in Tensorflow 2.1

2.1 Model Implementation

```
trainData, validData, testData, trainTarget, validTarget, testTarget =
↪  loadData()
model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(32, (3, 3), input_shape=(28, 28, 1),
        ↪  activation='relu'),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.MaxPooling2D((2,2), padding='SAME'),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(H, activation='relu'),
        tf.keras.layers.Dense(10, activation='softmax')
        ])
```

2.2 Model Training

The following trains the model, and plots cross-entropy and accuracy:

```
model.compile(optimizer='adam',
      loss='sparse_categorical_crossentropy',
      metrics=['sparse_categorical_crossentropy'])

history = model.fit(np.reshape(trainData, (len(trainData), 28, 28, 1)),
↪  trainTarget, batch_size = 35, epochs = 50)

plt.plot(history.history['loss'])
plt.plot(history.history['accuracy'])
```

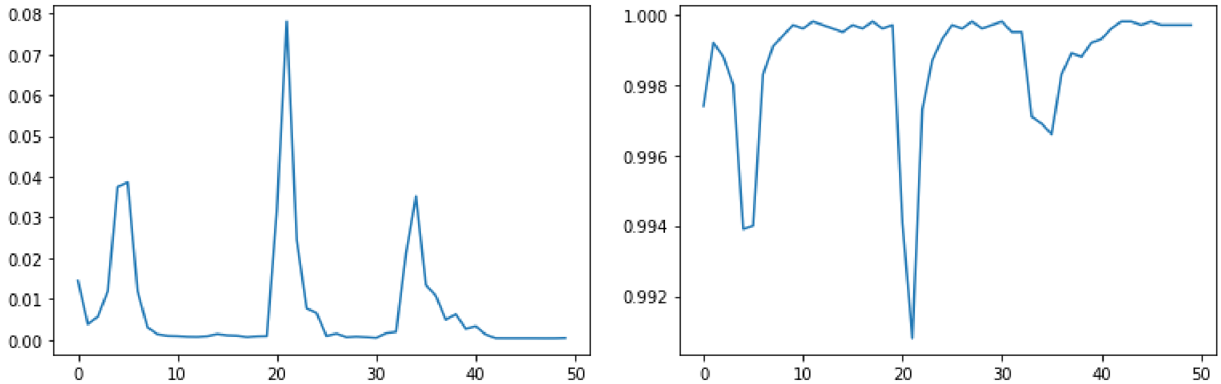

Figure 13: *Training Loss and Accuracy, using the tensorflow implementation of NN with batch size = 35, epochs = 50, and* $\lambda = 0$.

Final loss: 0.0000.
Final accuracy: 0.9999.

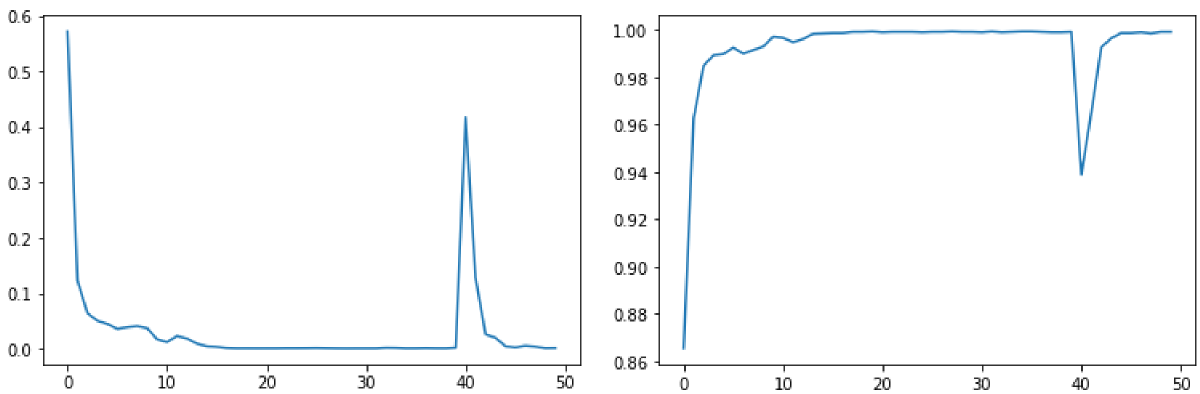

Figure 14: *Validation Loss and Accuracy, using the tensorflow implementation of NN with batch size = 0, epochs = 50, and* $\lambda = 0$.

Final loss: 0.0020.
Final accuracy: 0.9992.

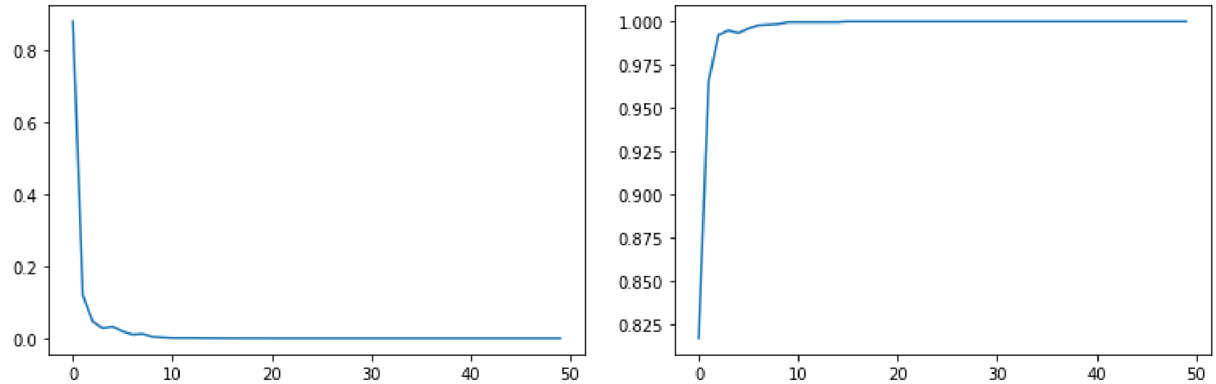

Figure 15: *Test Loss and Accuracy, using the tensorflow implementation of NN with batch size = 35, epochs = 50, and* $\lambda = 0$.

Final loss: 0.0000.
Final accuracy: 1.0000.

2.3 HyperParameter Investigation

The model was modified to incorporate regularization:

```
# add activity_regularizer = tf.keras.regularizers.l2(reg) to implement
↪  regularization, where reg is the regularization parameter

    model = tf.keras.Sequential([
            tf.keras.layers.Conv2D(32, (3, 3), input_shape=(28, 28, 1),
             ↪  activation='relu'),
            tf.keras.layers.BatchNormalization(),
            tf.keras.layers.MaxPooling2D((2,2), padding='SAME'),
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(H, activation='relu', activity_regularizer
             ↪  = tf.keras.regularizers.l2(reg)),
            tf.keras.layers.Dense(10, activation='softmax',
             ↪  activity_regularizer = tf.keras.regularizers.l2(reg))
            ])
```

Loss and accuracy when $\lambda = 0.01$.

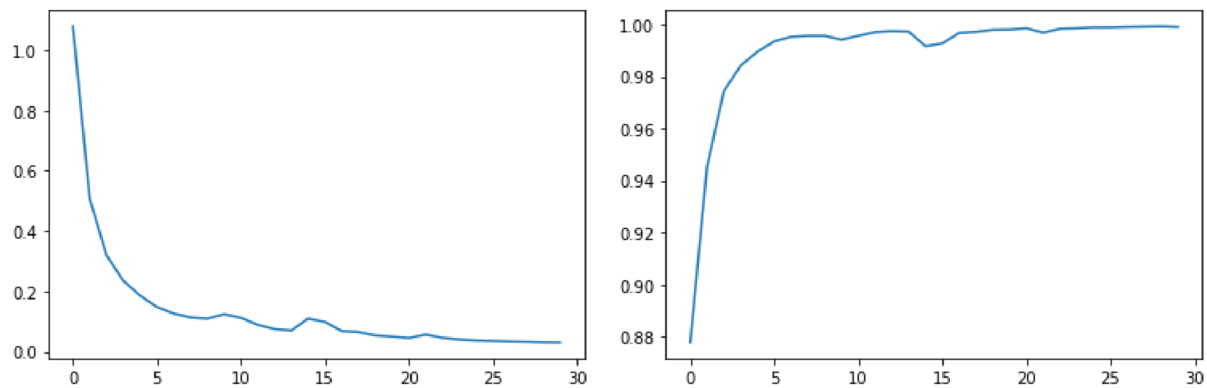

Figure 16: *Training Loss and Accuracy, using the tensorflow implementation of NN with batch size = 35, epochs = 50, and* $\lambda = 0.01$.

Final loss: 0.0298.
Final accuracy: 0.9991.

Loss and accuracy when $\lambda = 0.1$.

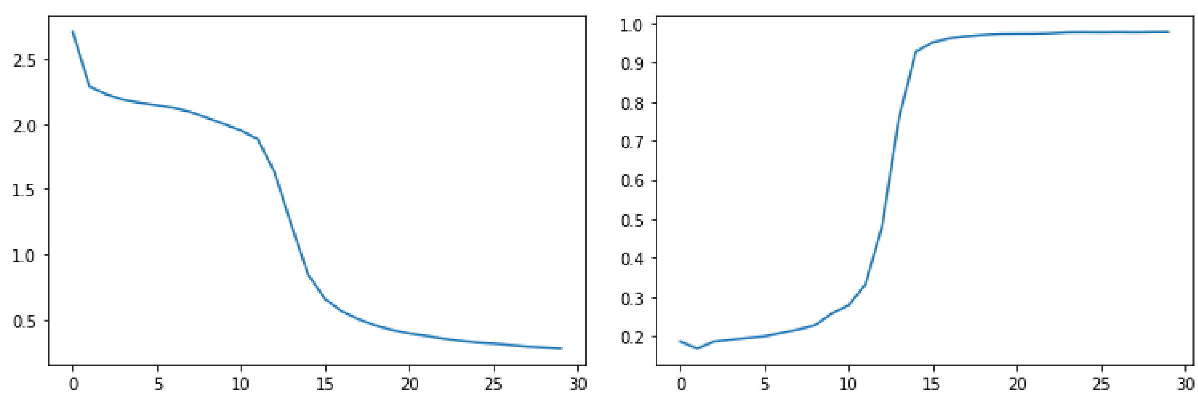

Figure 17: *Training Loss and Accuracy, using the tensorflow implementation of NN with batch size = 35, epochs = 50, and* $\lambda = 0.1$.

Final loss: 0.2795.
Final accuracy: 0.9782.

Loss and accuracy when $\lambda = 0.5$.

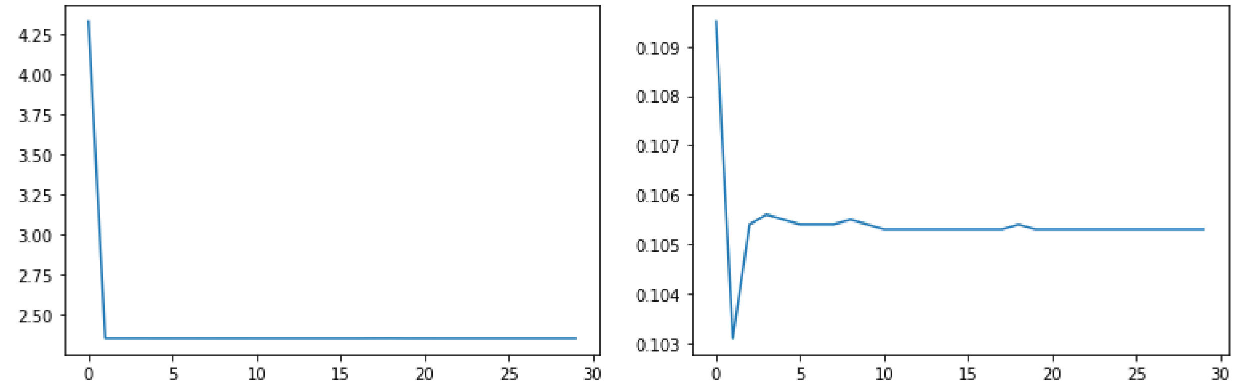

Figure 18: *Training Loss and Accuracy, using the tensorflow implementation of NN with batch size = 35, epochs = 50, and* $\lambda = 0.5$.

Final loss: 2.3525
Final accuracy: 0.1053.
The regularization parameter has become too large - the data is under fit.