Assignment1

April 2, 2020

1 Assignment 1 - TDT4265

Sara L. Ludvigsen and Emma H. Buøen

April 2020

1.1 Task 1 - Theory

1.1.1 1.a - Logistic Regression

The purpose of this task is to show that the gradient descent of the cost function is as given.

$$\frac{\partial C^n(w)}{\partial w_j} = -(y^n - \hat{y}^n)x_j^n$$

The cost function is as follows:

$$C(w) = -\frac{1}{N}\sum_{n=1}^{N} y^n \ln(\hat{y}^n) + (1-y^n)\ln(1-\hat{y}^n)$$

Where $\hat{y} = f_w(x) = \frac{1}{1+e^{-w^T x}}$ and $\frac{\partial f_w(x^n)}{\partial w_j} = x_j n f_w(x^n)(1 - f_w(x^n))$.

We simplify by splitting up the calculations:

$$\frac{\partial}{\partial w_j}\ln(\hat{y}^n) = \frac{\partial}{\partial w_j}\ln(f_w(x^n)) = \frac{1}{f_w(x^n)}\frac{\partial}{\partial w_j}f_w(x^n) = \frac{1}{f_w(x^n)}x_j^n f_w(x^n)(1-f_w(x^n)) = x_j^n(1-f_w(x^n))$$

$$\begin{aligned}\frac{\partial}{\partial w_j}\ln(1-\hat{y}^n) &= \frac{\partial}{\partial w_j}\ln(1-f_w(x^n)) = \frac{1}{1-f_w(x^n)}\frac{\partial}{\partial w_j}(1-f_w(x^n)) \\ &= \frac{-1}{1-f_w(x^n)}x_j^n f_w(x^n)(1-f_w(x^n)) = -x_j^n f_w(x^n)\end{aligned}$$

With these calculations, we can show that:

$$\frac{\partial}{\partial w_j}C^n(w) = -\frac{\partial}{\partial w_j}(y^n\ln(\hat{y}^n) + (1-y^n)\ln(1-\hat{y}^n)) = -y^n x_j^n(1-f_w(x^n)) + (1-y^n)x_j^n f_w(x^n) = \underline{-(y^n-\hat{y}^n)x_j^n}$$

1.1.2 1.b - Softmax Regression

In this task we derive the gradient for Softmax Regression. Our goal is to show that

$$\frac{\partial}{\partial w_{kj}} C^n(w) = -\frac{1}{K} x_j^n (y_k^n - \hat{y}_k^n).$$

The cost function is:

$$C(w) = -\frac{1}{N \cdot K} \sum_{n=1}^{N} \sum_{k=1}^{K} y_k^n \ln(\hat{y}_k^n)$$

Where $\hat{y}_k^n = \frac{e^{z_k^n}}{\sum_{k'=1}^{K} e^{z_{k'}^n}}$ and $z_k^n = w_k^T x^n$.

$$\sum_{k=1}^{K} y_k^n = 1$$

$$\ln(\frac{a}{b}) = \ln(a) - \ln(b)$$

$$\frac{\partial}{\partial w_{kj}} C^n(w) = \frac{\partial C^n(w)}{\partial \hat{y}_k^n} \frac{\partial \hat{y}_k^n}{\partial z_k^n} \frac{\partial z_k^n}{\partial w_{kj}}$$

$$\frac{\partial z_k^n}{\partial w_{kj}} = x^n$$

When $k = i$:

$$\frac{\partial \hat{y}_k^n}{\partial z_k^n} = \frac{e^{z_k^n}}{\sum_{k'=1}^{K} e^{z_{k'}^n}} - \frac{e^{z_k^n} \cdot e^{z_k^n}}{(\sum_{k'=1}^{K} e^{z_{k'}^n})^2} = \hat{y}_k^n (1 - \hat{y}_k^n)$$

When $k \neq i$:

$$\frac{\partial \hat{y}_i^n}{\partial z_k^n} = -\frac{e^{z_i^n} \cdot e^{z_k^n}}{(\sum_i e^{z_k^n})^2} = -\hat{y}_k^n \cdot \hat{y}_i^n$$

$$\frac{\partial C^n(w)}{\partial \hat{y}_i^n} = -\frac{1}{K} y_i^n \frac{1}{(\hat{y}_i^n)}$$

This leads to

$$\frac{\partial}{\partial w_{kj}} C^n(w) = \frac{-1}{K} (\sum_i y_i^n \frac{1}{(\hat{y}_i^n)} (-\hat{y}_k^n \cdot \hat{y}_i^n) x^n + y_k^n \frac{1}{\hat{y}_k^n} \hat{y}_k^n x_j^n) = \frac{-1}{K} ((\underbrace{\sum_i y_i^n}_{=1})(-\hat{y}_k^n x_j^n) + y_k^n x_j^n) = \underline{\frac{-1}{K} x_j^n (y_k^n - \hat{y}_k^n)}$$

1.2 Task 2 - Logistic Regression through Gradient Descent

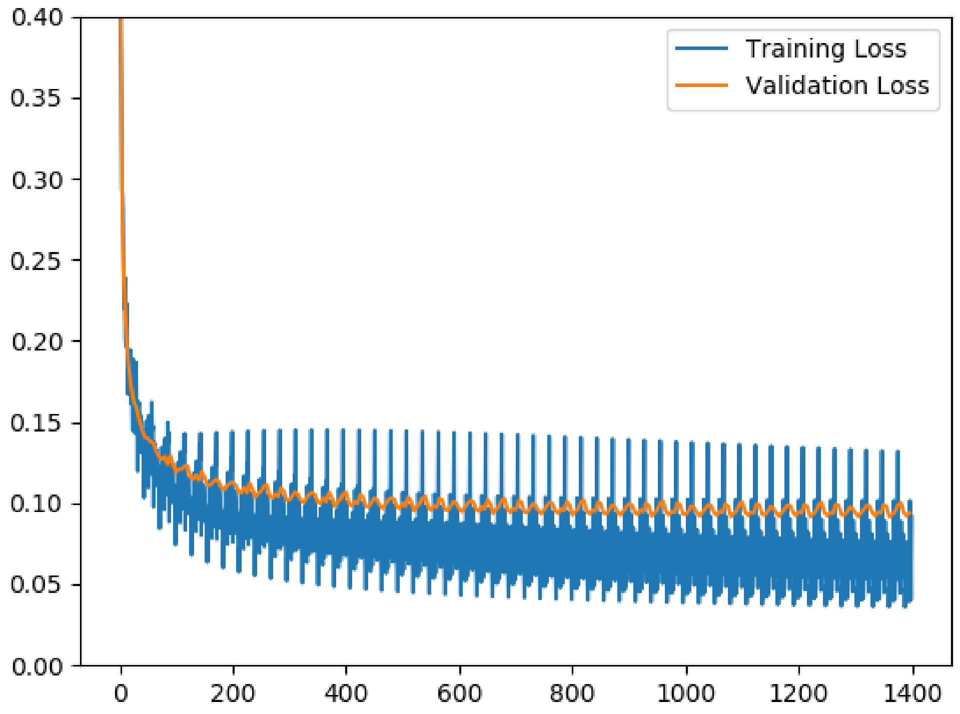
1.2.1 2.a

```
[ ]: def pre_process_images(X: np.ndarray):
         """
```

```
    Args:
        X: images of shape [batch size, 784] in the range (0, 255)
    Returns:
        X: images of shape [batch size, 785] in the range (0, 1)
    """
    assert X.shape[1] == 784,\
        f"X.shape[1]: {X.shape[1]}, should be 784"

    X = np.divide(X,255)
    X = np.insert(X, 0, 1, axis=1)
    return X

class BinaryModel:

    def __init__(self, l2_reg_lambda: float):
        # Define number of input nodes
        self.I = 785
        self.w = np.zeros((self.I, 1))
        self.grad = None

        # Hyperparameter for task 3
        self.l2_reg_lambda = l2_reg_lambda

    def forward(self, X: np.ndarray) -> np.ndarray:
        """
        Args:
            X: images of shape [batch size, 785]
        Returns:
            y: output of model with shape [batch size, 1]
        """
        # Sigmoid
        size = X.shape[0]
        y = np.ones((size,1))
        temp = np.dot(self.w.transpose(), X.transpose()).transpose()

        for index in range(0, len(y)):
            y[index] = 1/(1 + math.exp(-1*temp[index]))
        return y

    def backward(self, X: np.ndarray, outputs: np.ndarray, targets: np.ndarray)␣
 ↪-> None:
        """
        Args:
            X: images of shape [batch size, 785]
            outputs: outputs of model of shape: [batch size, 1]
            targets: labels/targets of each image of shape: [batch size, 1]
        """
        assert targets.shape == outputs.shape,\
            f"Output shape: {outputs.shape}, targets: {targets.shape}"
        self.grad = np.zeros_like(self.w)
        assert self.grad.shape == self.w.shape,\
            f"Grad shape: {self.grad.shape}, w: {self.w.shape}"
        y_hat = outputs
        y = targets
        self.grad = -np.dot(X.transpose(), y - y_hat)/(X.shape[0])

    def zero_grad(self) -> None:
        self.grad = None

def cross_entropy_loss(targets: np.ndarray, outputs: np.ndarray) -> float:
    """
    Args:
        targets: labels/targets of each image of shape: [batch size, 1]
        outputs: outputs of model of shape: [batch size, 1]
    Returns:
        Cross entropy error (float)
    """
    assert targets.shape == outputs.shape,\
        f"Targets shape: {targets.shape}, outputs: {outputs.shape}"

    y_hat = outputs
    y = targets
    N = y_hat.shape[0]
    c = 0
    for n in range(0,N):
        c += y[n]*math.log(y_hat[n]) + (1-y[n])*math.log(1-y_hat[n])
    c = -1/N*c
    return c
```

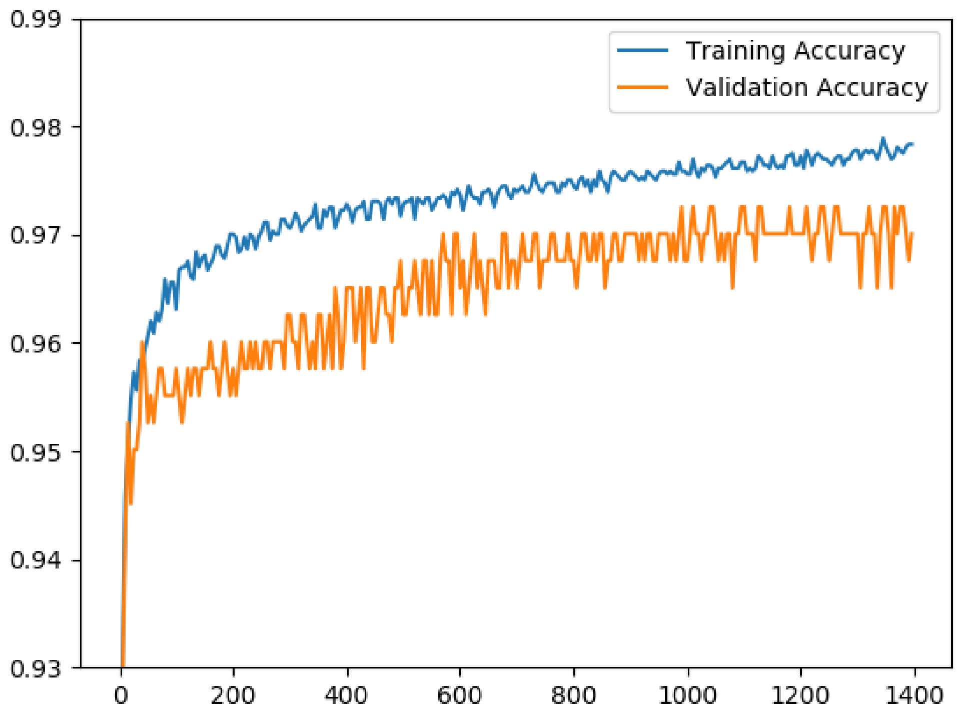
1.2.2 2.b

The figure below is the plot of the loss of the training and the validation sets. The loss is shown on the y-axis and the gradient step is on the x-axis.



1.2.3 2.c

The figure below shows the accuracy of our model. The accuracy is on the y-axis and the gradient step is on the x-axis.



```
[ ]: # Code for Task 2.b and 2.c

def calculate_accuracy(X: np.ndarray, targets: np.ndarray, model: BinaryModel)␣
 ↪-> float:
    """
    Args:
        X: images of shape [batch size, 785]
        targets: labels/targets of each image of shape: [batch size, 1]
        model: model of class BinaryModel
    Returns:
        Accuracy (float)
    """
    number_of_predictions = X.shape[0]
    number_of_rights = 0
    y_hat = model.forward(X)

    for index in range (0, number_of_predictions):
        if y_hat[index] >= 0.5:
            y_hat[index] = 1
        else:
            y_hat[index] = 0

        if y_hat[index] == targets[index]:
                number_of_rights += 1
    # Task 2c
    accuracy = number_of_rights/number_of_predictions
    return accuracy


def train(
        num_epochs: int,
        learning_rate: float,
        batch_size: int,
        l2_reg_lambda: float # Task 3 hyperparameter. Can be ignored before this.
        ):
    """
        Function that implements logistic regression through mini-batch
        gradient descent for the given hyperparameters
    """
    global X_train, X_val, X_test
    # Utility variables
    num_batches_per_epoch = X_train.shape[0] // batch_size
    num_steps_per_val = num_batches_per_epoch // 5
    train_loss = {}
    val_loss = {}
    train_accuracy = {}
    val_accuracy = {}
    model = BinaryModel(l2_reg_lambda)
    if X_train.shape[1]==784:
        X_train = pre_process_images(X_train)
    if X_test.shape[1]==784:
        X_test = pre_process_images(X_test)
    if X_val.shape[1]==784:
        X_val = pre_process_images(X_val)

    global_step = 0
    for epoch in range(num_epochs):
        for step in range(num_batches_per_epoch):
            # Select our mini-batch of images / labels
            start = step * batch_size
            end = start + batch_size
            X_batch, Y_batch = X_train[start:end], Y_train[start:end]

            y_hat = model.forward(X_batch)

            model.backward(X_batch, y_hat, Y_batch)
            model.w += -1*learning_rate*model.grad

            # Track training loss continuously
            _train_loss = cross_entropy_loss(Y_batch, y_hat)
            train_loss[global_step] = _train_loss
            # Track validation loss / accuracy every time we progress 20%␣
↪through the dataset
            if global_step % num_steps_per_val == 0:
                _val_loss = cross_entropy_loss(Y_val, model.forward(X_val))
                val_loss[global_step] = _val_loss

                train_accuracy[global_step] = calculate_accuracy(
                    X_train, Y_train, model)
                val_accuracy[global_step] = calculate_accuracy(
                    X_val, Y_val, model)

            global_step += 1
    return model, train_loss, val_loss, train_accuracy, val_accuracy
```

1.2.4 2.d

From the plot in task 2.b we can see that our graph is very noisy. This made it difficult for us to know when to stop training. In order to avoid overfitting of the model when training, the training should be stopped when the validation loss is increasing. We tried to implement an ealry stopping algorithm that stops the training when the validation loss increased five times in a row, after a specified time of training. Due to the noise in our graph, this never happened. We think we would have seen better results if we ran our data through a filter. Unfortunately, we encountered some problems when trying to do this, and we ran out of time.

In the plot from 2.b we observe that there is no longer an improvement in the validation loss after about 400 gradient steps.

[ ]:
```
## Early stopping

def train(
        num_epochs: int,
        learning_rate: float,
        batch_size: int,
        l2_reg_lambda: float # Task 3 hyperparameter. Can be ignored before this.
        ):
    """
        Function that implements logistic regression through mini-batch
        gradient descent for the given hyperparameters
    """
    global X_train, X_val, X_test
    # Utility variables
    num_batches_per_epoch = X_train.shape[0] // batch_size
    num_steps_per_val = num_batches_per_epoch // 5
    train_loss = {}
    val_loss = {}
    train_accuracy = {}
    val_accuracy = {}
    model = BinaryModel(l2_reg_lambda)
    if X_train.shape[1]==784:
        X_train = pre_process_images(X_train)
    if X_test.shape[1]==784:
        X_test = pre_process_images(X_test)
    if X_val.shape[1]==784:
        X_val = pre_process_images(X_val)

    global_step = 0
    global_count = 0
    for epoch in range(num_epochs):
        for step in range(num_batches_per_epoch):
            # Select our mini-batch of images / labels
            start = step * batch_size
            end = start + batch_size
            X_batch, Y_batch = X_train[start:end], Y_train[start:end]

            y_hat = model.forward(X_batch)

            model.backward(X_batch, y_hat, Y_batch)
            model.w += -1*learning_rate*model.grad

            # Track training loss continuously
            _train_loss = cross_entropy_loss(Y_batch, y_hat)
            train_loss[global_step] = _train_loss
            # Track validation loss / accuracy every time we progress 20%␣
↪through the dataset

            if global_step % num_steps_per_val == 0:
                _val_loss = cross_entropy_loss(Y_val, model.forward(X_val))
                val_loss[global_step] = _val_loss

                if _val_loss < val_loss[global_step-(num_batches_per_epoch //␣
↪num_steps_per_val)]:
                    count += 1
                    if global_count > 3:
                        train_accuracy[global_step] =␣
↪calculate_accuracy(X_train, Y_train, model)
                        val_accuracy[global_step] = calculate_accuracy(X_val,␣
↪Y_val, model)
                        print("EARLY STOP")
                        return model, train_loss, val_loss, train_accuracy,␣
↪val_accuracy
                else: global_count = 0
                train_accuracy[global_step] = calculate_accuracy(
                    X_train, Y_train, model)
                val_accuracy[global_step] = calculate_accuracy(
                    X_val, Y_val, model)

            global_step += 1
    return model, train_loss, val_loss, train_accuracy, val_accuracy
```

1.3 Task 3 - Regularization

1.3.1 3.a

The new cost function is $ J(w) = C(w) + $\lambda R(w)\$ where R(w)$ is the complexity penalty and λ is the strength of the regularization.

From the problem description, it is known that

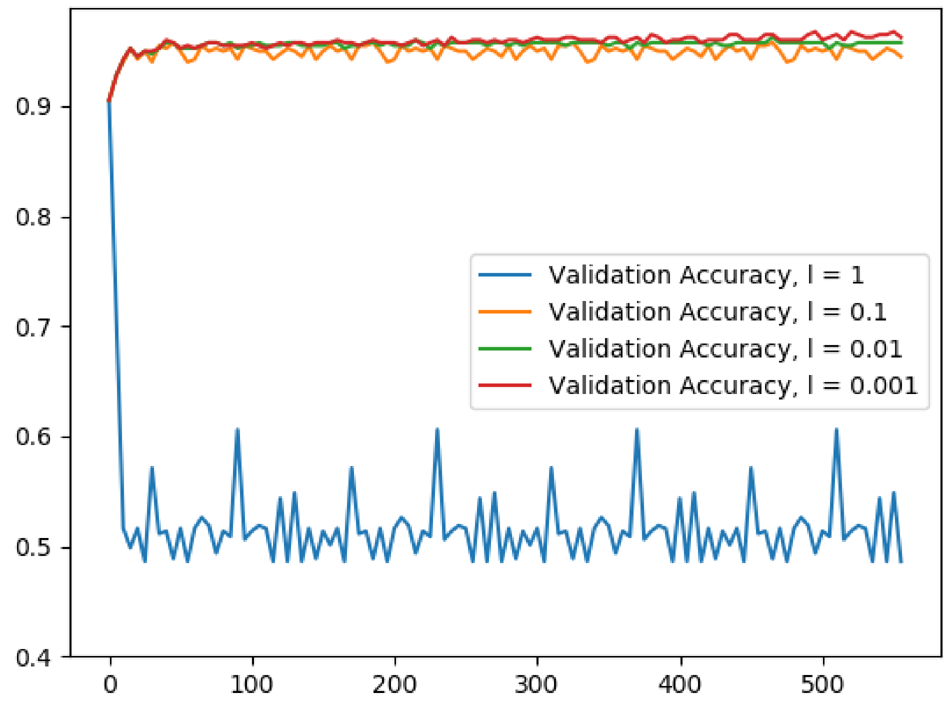
$$R(w) = \|w\|^2 = \sum_{i,j} w_{i,j}^2 = \sum_{i=1}^{I}\sum_{k=1}^{K} w_{i,j}^2.$$

We also know that $I = 785$ and $K = 1$.

We want to find the new gradient of the cost function, in other words, we need to find $\frac{\partial}{\partial w}R(w)$.

$$\frac{\partial}{\partial w}R(w) = \sum_{i=1}^{I}\sum_{k=1}^{K} w_{i,j}^2 \frac{\partial}{\partial w} w_{i,j} = \sum_{i=1}^{I} 2w_i = 2\sum_{i=1}^{I} w_i$$

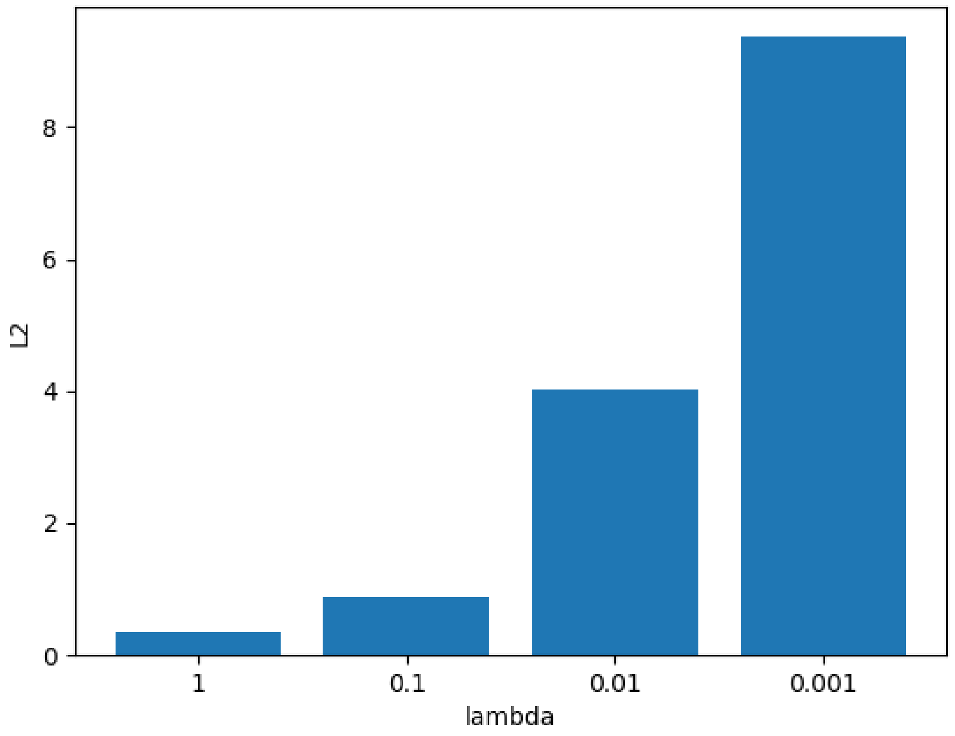
1.3.2 3.b



We observe that when lambda is 1, the penalty for a complex model is too high, and our model is not rewarded enough for correct predictions. It is therefore no better than guessing.

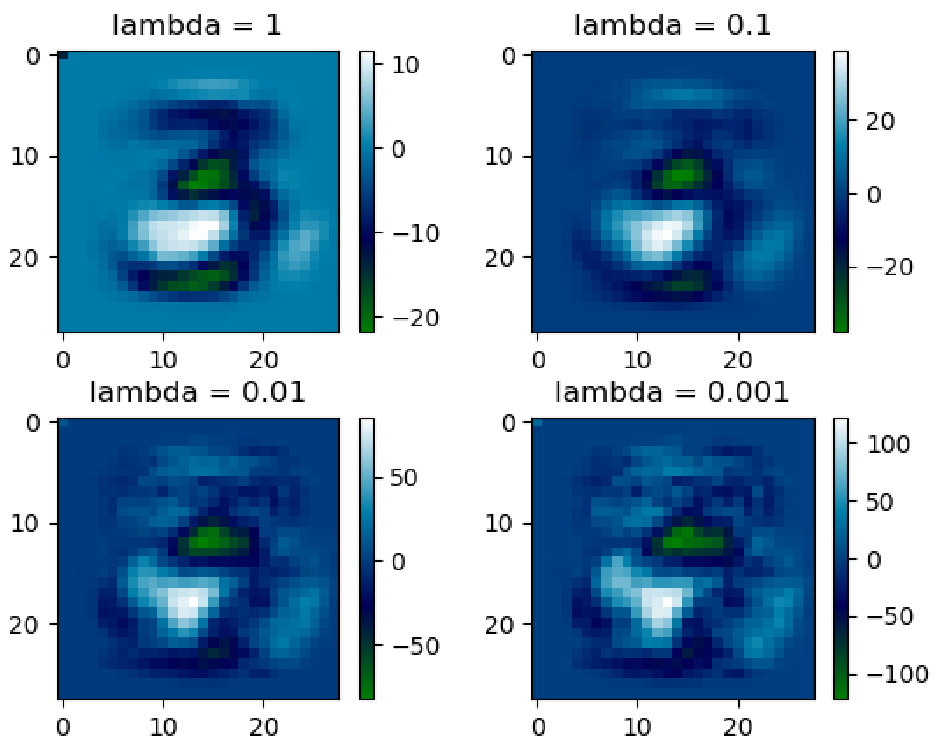
As expected, we are able to achieve the highest level of accuracy with the most complex model.

1.3.3 3.c



The L2-norm is increasing with the complexity of the model.

1.3.4 3.d



We clearly see that the lower lambda we use, the more noise is included when calculating the weights. The higher values of lambda, on the other hand, does not include specific areas in their calculations. The models with higher values of lambda will have difficulty recognizing variations of the same number. The range is lower when lambda is higher.

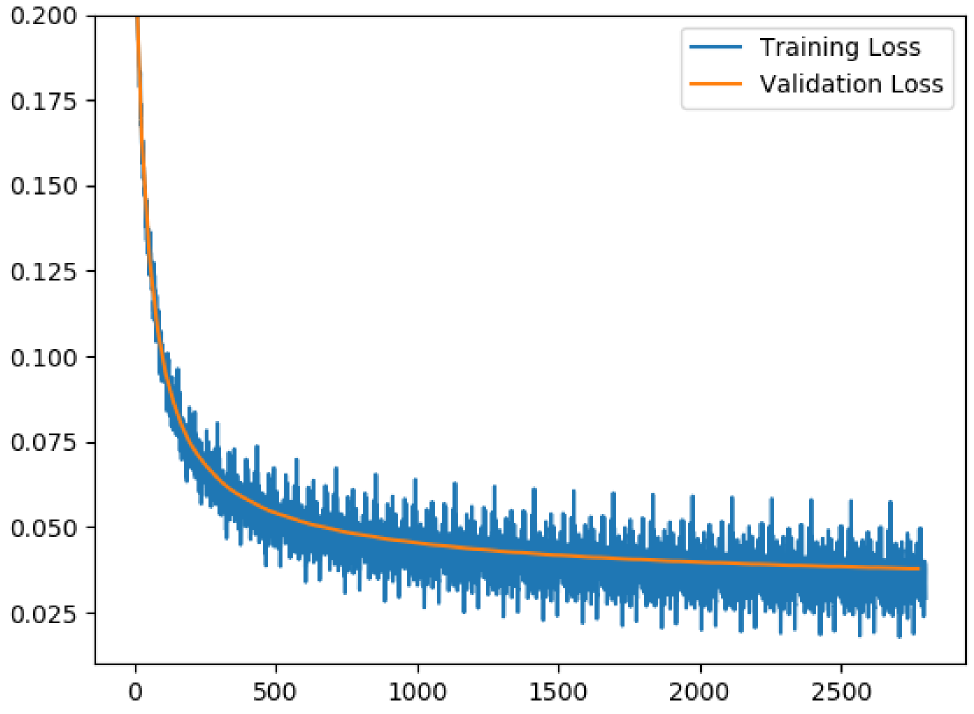
1.4 Task 4

1.4.1 4.a

```
[ ]: def cross_entropy_loss(targets: np.ndarray, outputs: np.ndarray):
         """
         Args:
             targets: labels/targets of each image of shape: [batch size, num_classes]
             outputs: outputs of model of shape: [batch size, num_classes]
         Returns:
             Cross entropy error (float)
         """
         assert targets.shape == outputs.shape,\
             f"Targets shape: {targets.shape}, outputs: {outputs.shape}"

         ce = targets * np.log(outputs)
         ce = np.sum(ce, axis=0)
         ce = np.sum(ce)
```

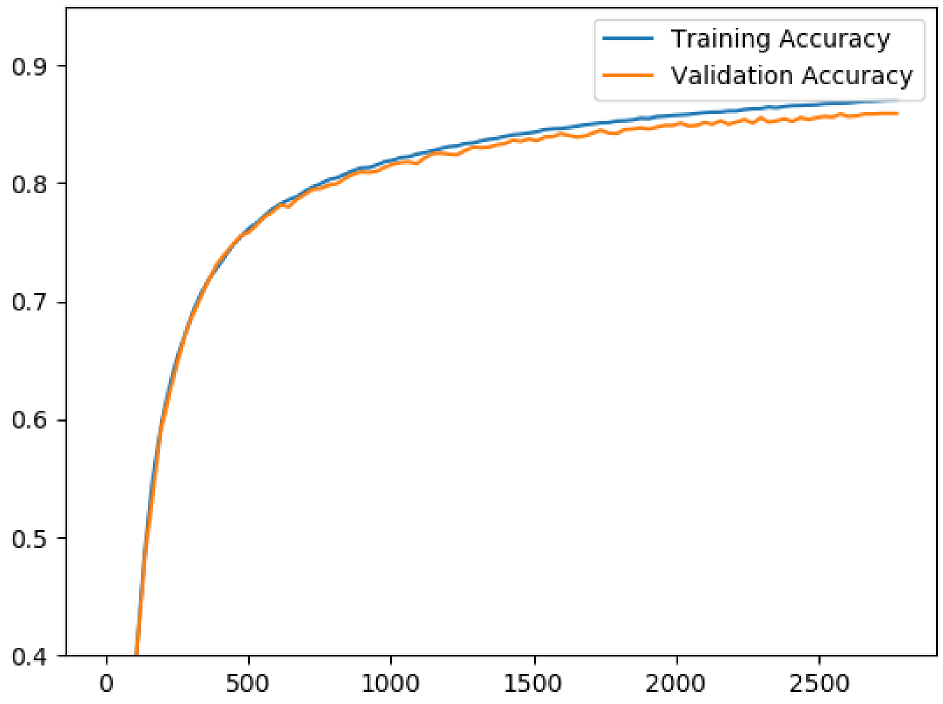
```
    ce = -(1/(targets.shape[0]*targets.shape[1]))*ce
    return ce


class SoftmaxModel:

    def __init__(self, l2_reg_lambda: float):
        # Define number of input nodes
        self.I = 785

        # Define number of output nodes
        self.num_outputs = 10
        self.w = np.zeros((self.I, self.num_outputs))
        self.grad = None

        self.l2_reg_lambda = l2_reg_lambda

    def forward(self, X: np.ndarray) -> np.ndarray:
        """
        Args:
            X: images of shape [batch size, 785]
        Returns:
            y: output of model with shape [batch size, num_outputs]
        """
        Z = np.array(0)
        Z = np.matmul(X, self.w).transpose() # Z.shape = (num_outputs, batch␣
↪size)

        eZ = np.array(Z.shape)
        eZ = np.exp(Z)
        test = np.sum(eZ, axis=0)
        eZ /= np.sum(eZ, axis=0, keepdims=True)
        output = eZ.transpose() #eZ.shape = (batch size, num_outputs)
        return output

    def backward(self, X: np.ndarray, outputs: np.ndarray, targets: np.ndarray)␣
↪-> None:
        """
        Args:
            X: images of shape [batch size, 785]
            outputs: outputs of model of shape: [batch size, num_outputs]
            targets: labels/targets of each image of shape: [batch size,␣
↪num_classes]
        """
        assert targets.shape == outputs.shape,\
            f"Output shape: {outputs.shape}, targets: {targets.shape}"
        self.grad = np.zeros_like(self.w)
        assert self.grad.shape == self.w.shape,\
             f"Grad shape: {self.grad.shape}, w: {self.w.shape}"

        Z = np.array(0)
        Z = np.matmul(X.transpose(),(targets - outputs)) #Z.shape = (785,␣
 ↪num_outputs)

        self.grad = - Z  / (targets.shape[0] * targets.shape[1]) + 2*self.
 ↪l2_reg_lambda*self.w

    def zero_grad(self) -> None:
        self.grad = None


def one_hot_encode(Y: np.ndarray, num_classes: int):
    """
    Args:
        Y: shape [Num examples, 1]
        num_classes: Number of classes to use for one-hot encoding
    Returns:
        Y: shape [Num examples, num classes]
    """
    encoded = np.zeros((Y.size, num_classes))
    encoded[np.arange(Y.size), Y.squeeze()] = 1
    return encoded


def gradient_approximation_test(model: SoftmaxModel, X: np.ndarray, Y: np.
 ↪ndarray):
    """
        Numerical approximation for gradients. Should not be edited.
        Details about this test is given in the appendix in the assignment.
    """
    epsilon = 1e-2
    for i in range(model.w.shape[0]):
        for j in range(model.w.shape[1]):
            orig = model.w[i, j].copy()
            model.w[i, j] = orig + epsilon
            logits = model.forward(X)
            cost1 = cross_entropy_loss(Y, logits)
            model.w[i, j] = orig - epsilon
            logits = model.forward(X)
            cost2 = cross_entropy_loss(Y, logits)
            gradient_approximation = (cost1 - cost2) / (2 * epsilon)
            model.w[i, j] = orig
            # Actual gradient
            logits = model.forward(X)
            model.backward(X, logits, Y)
            difference = gradient_approximation - model.grad[i, j]
            assert abs(difference) <= epsilon**2,\
                f"Calculated gradient is incorrect. " \
                f"Approximation: {gradient_approximation}, actual gradient:␣
↪{model.grad[i, j]}\n" \
                f"If this test fails there could be errors in your cross entropy␣
↪loss function, " \
                f"forward function or backward function"
```

1.4.2 4.b



The figure shows the training and validation loss for the softmax regression.

1.4.3 4.c



The figure shows the training and validation accuracy for the softmax regression.

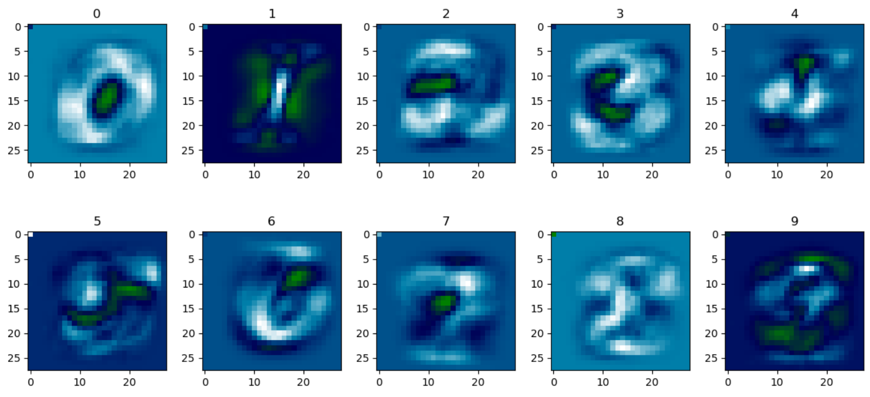
The final accuracy and loss values were as follows:

Final Metric	Value
Final Train Cross Entropy Loss	0.03429192881141587
Final Test Entropy Loss	0.037827305981061383
Final Validation Cross Entropy Loss	0.037827305981061383
Final Train accuracy	0.8712222222222222
Final Validation accuracy	0.8615
Final Test accuracy	0.917

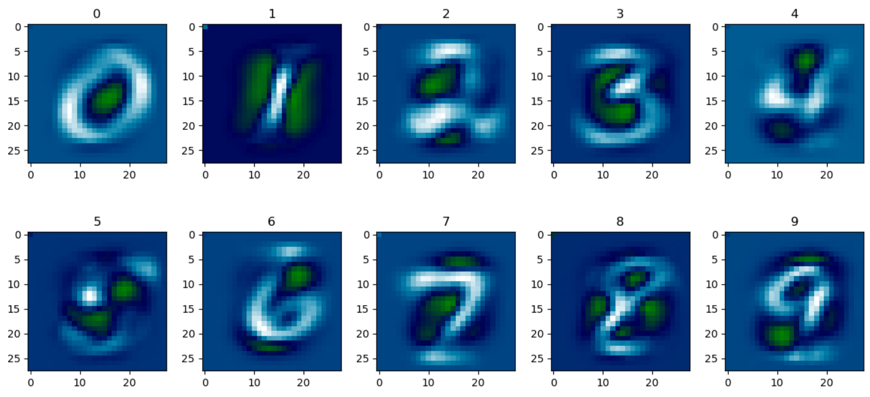
1.4.4 4.d

```
[ ]: self.grad = - Z  / (targets.shape[0] * targets.shape[1]) + 2*self.
     ↪l2_reg_lambda*self.w
```

1.4.5 4.e



Lambda = 0.0:



Lambda = 0.1:

When lambda = 0.1, the cost function penalizes comlpexity. This is why the weights for the models are less noisy. Without l2-regularization, i.o.w when lambda = 0, the model can be as complex as it wants to, without being penalized for it. This is why its weights are more noisy.

[ ]:
```
# Plot weight image

model, train_loss, val_loss, train_accuracy, val_accuracy = train(
    num_epochs=num_epochs,
    learning_rate=learning_rate,
    batch_size=batch_size,
    l2_reg_lambda=l2_reg_lambda)

#Plot weight image
fig, axs = plt.subplots(2,5, figsize=(15, 6), facecolor='w', edgecolor='k')
fig.subplots_adjust(hspace = .5, wspace=.001)
axs = axs.ravel()
for i in range(10):
    axs[i].imshow(255*(model.w[0:-1,i].reshape(28,28)),cmap="ocean")
    axs[i].set_title(str(i))
plt.savefig("task4d_lambda_0.0.png")
plt.show()
```