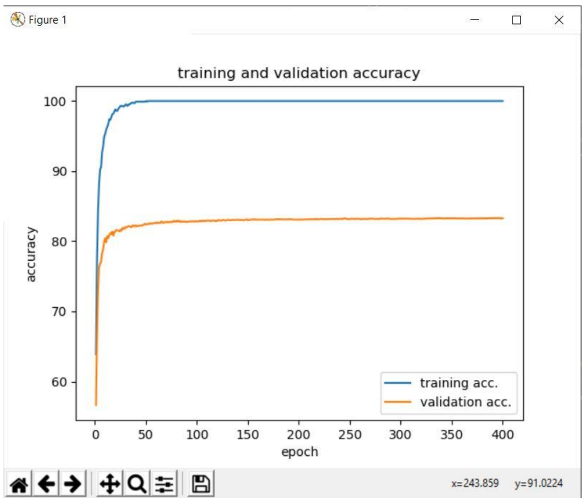
Neural Networks and Deep Learning – Summer Term 2019

Team - 09	
Name: Rajib Chandra Das Matriculation Number: 1140657 Email: stu218517@mail.uni-kiel.de	Name: M M Mahmudul Hassan Matriculation Number: 1140658 Email: stu218518@mail.uni-kiel.de

Exercise sheet 5

Exercise 1 (Overfitting):

Using the validation data as evaluation data along with changing the learning rate and the number of batch we are having the following graphs:



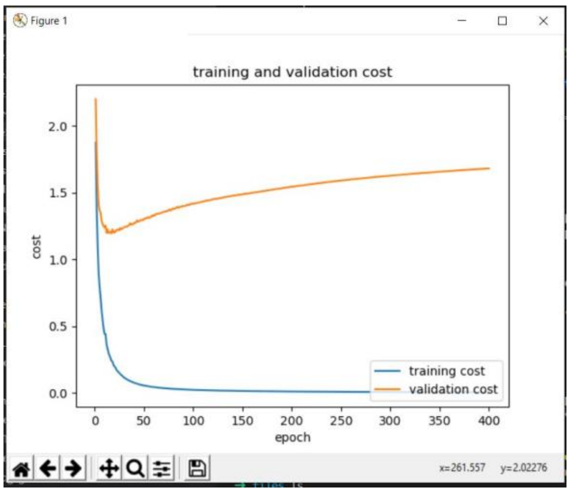
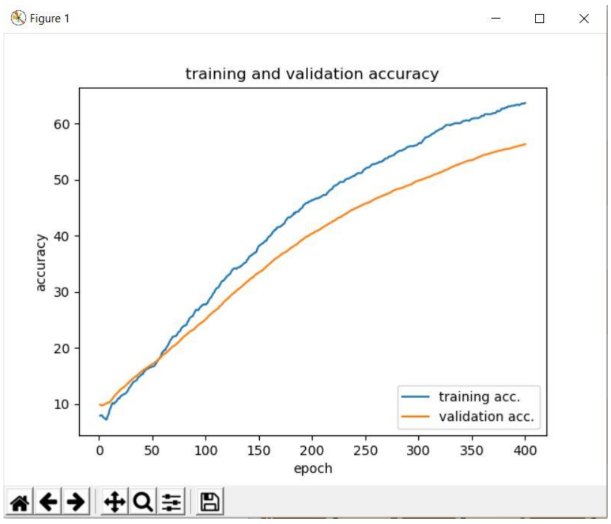


Fig: learning rate = 0.5, batch = 10



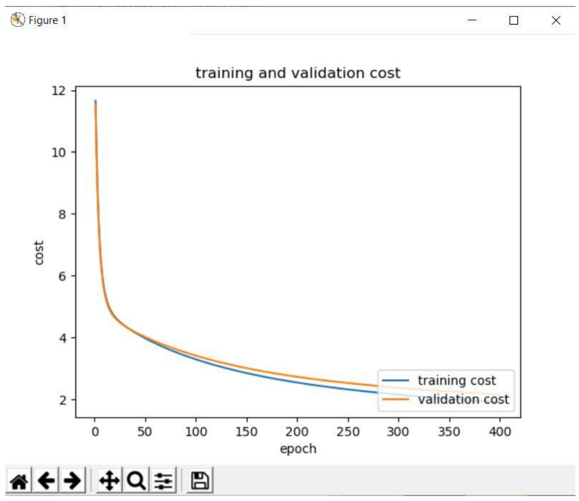
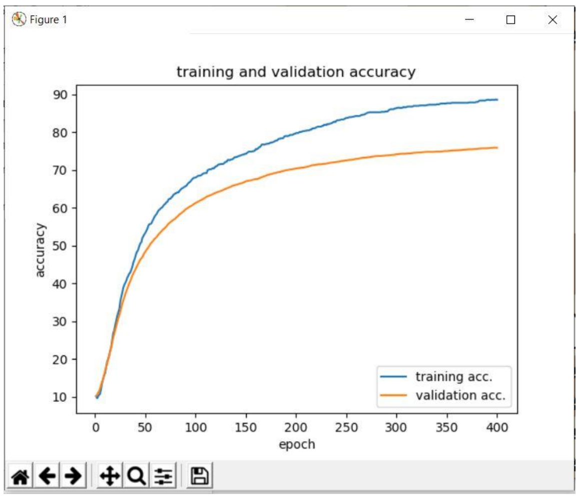


Fig: learning rate = 0.001, batch = 10



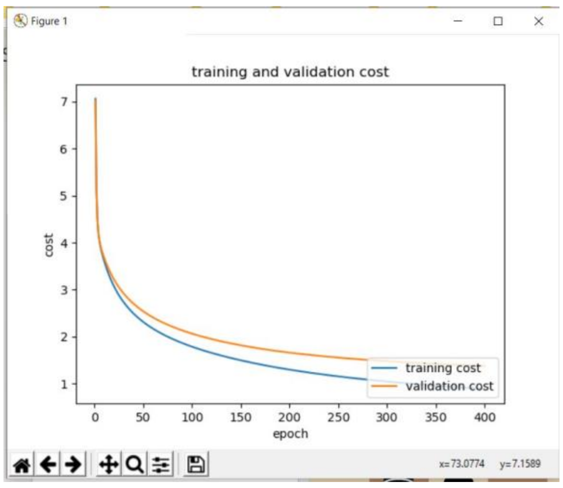
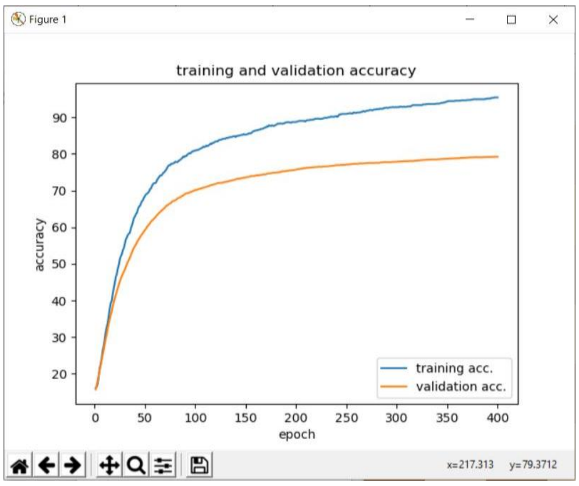


Fig: learning rate = 0.005, batch = 10



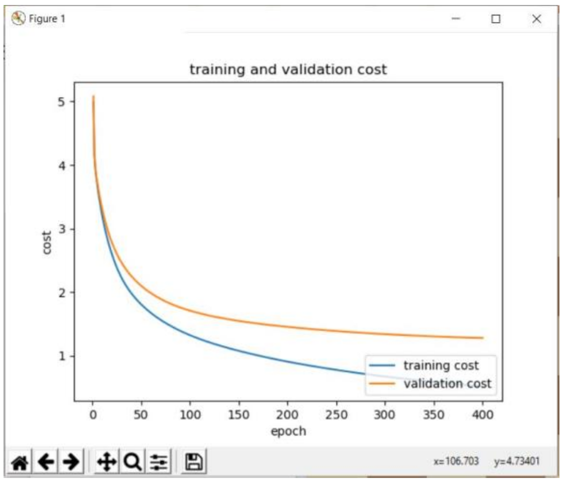
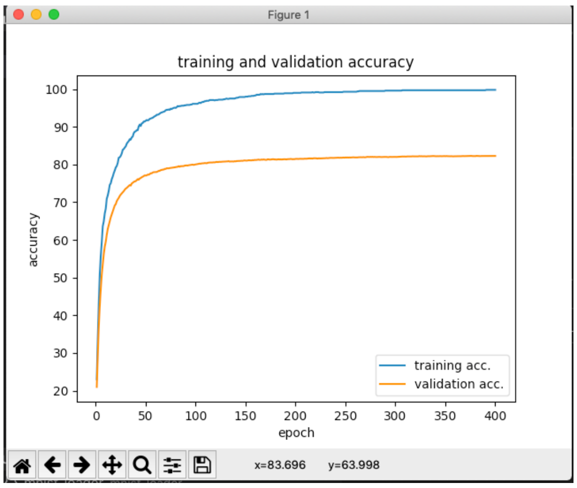


Fig: learning rate = 0.01, batch = 10



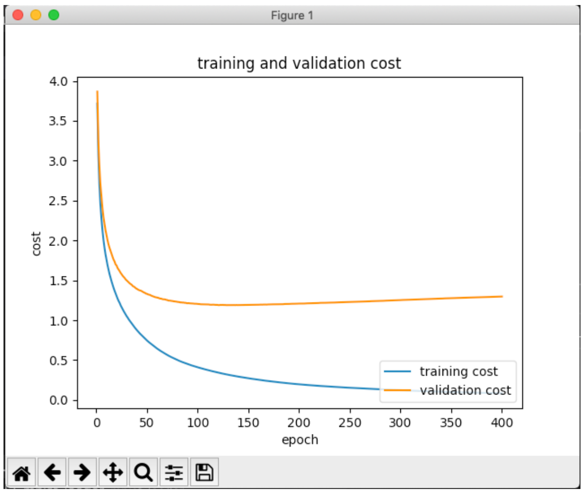
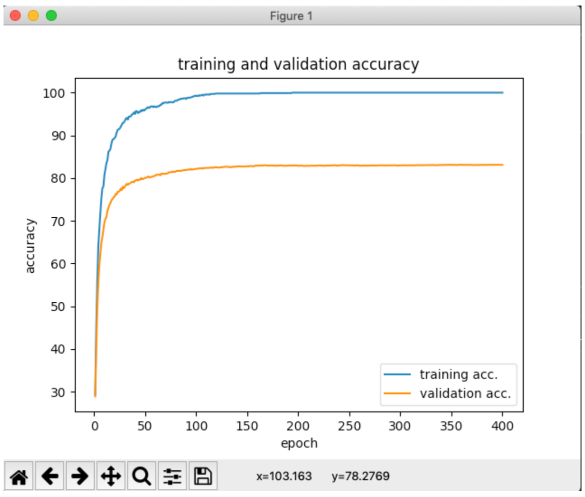


Fig: learning rate = 0.05, batch = 10



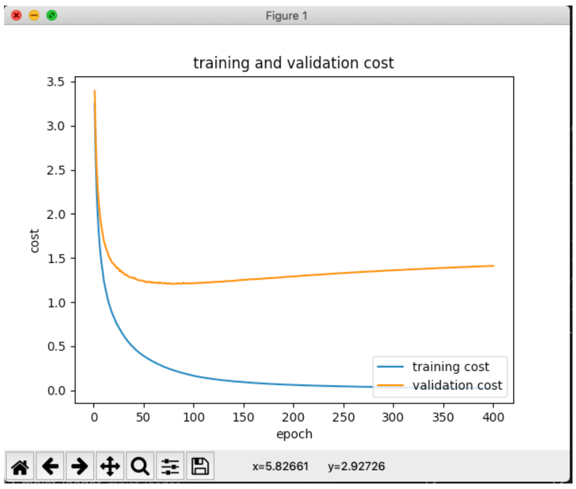
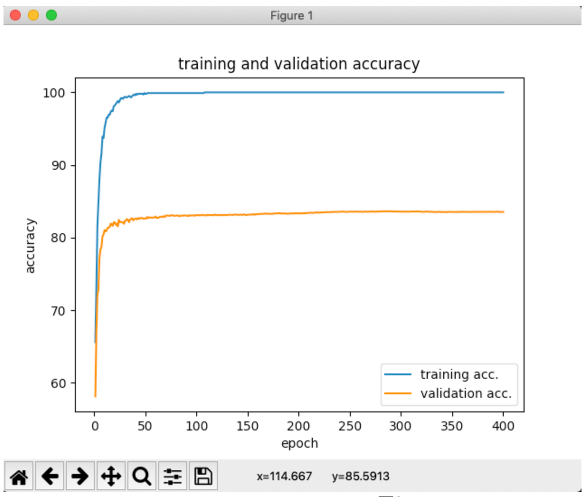


Fig: learning rate = 0.1, batch = 10



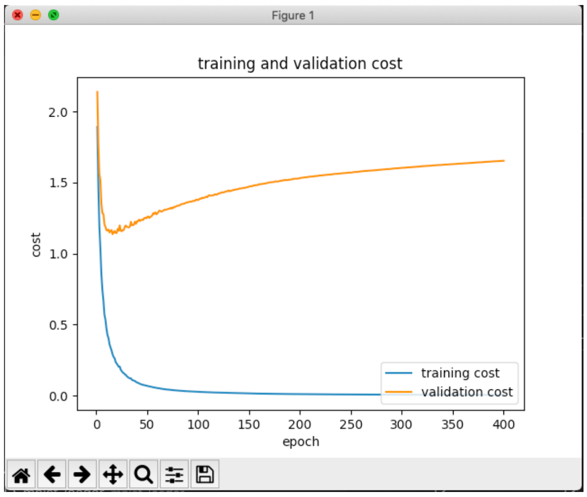
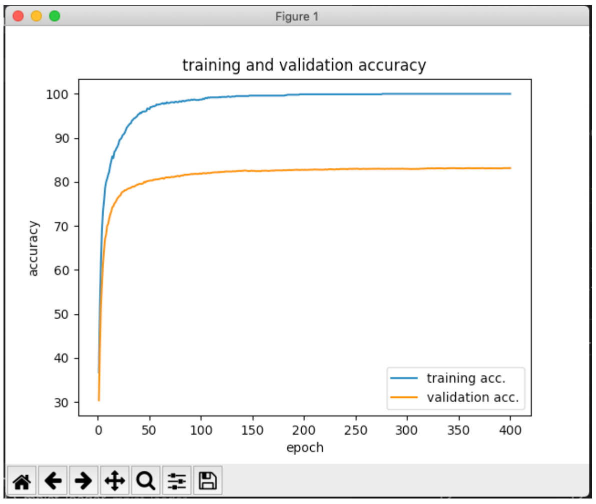


Fig: learning rate = 0.05, batch = 1



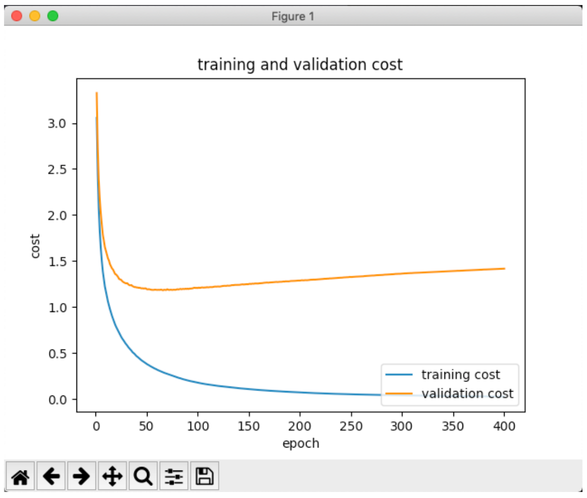
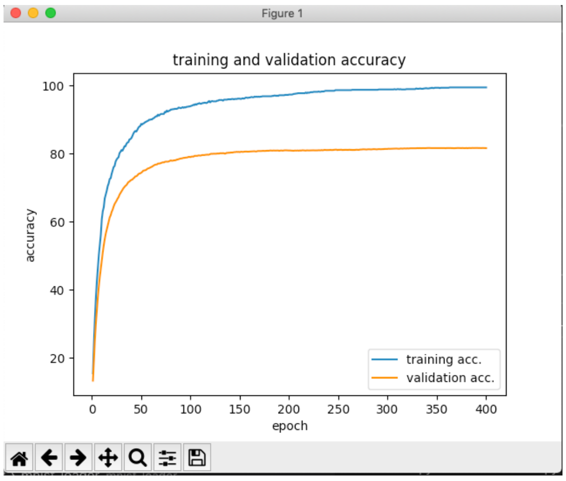


Fig: learning rate = 0.05, batch = 5



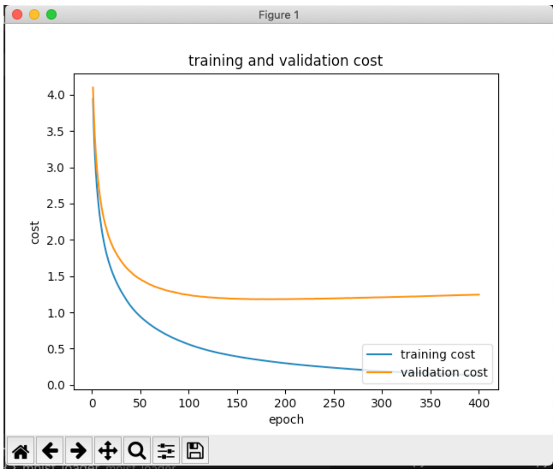
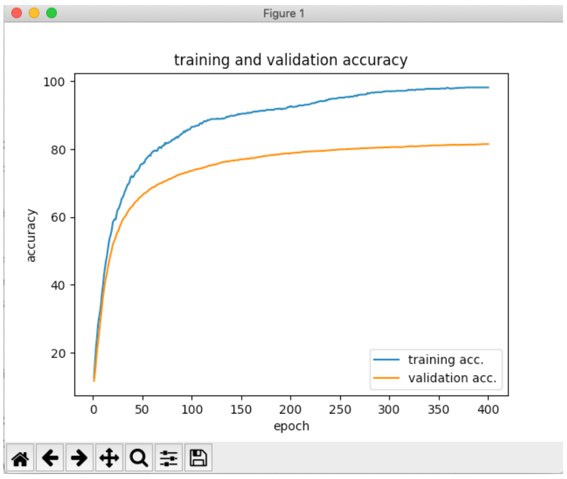


Fig: learning rate = 0.05, batch = 15



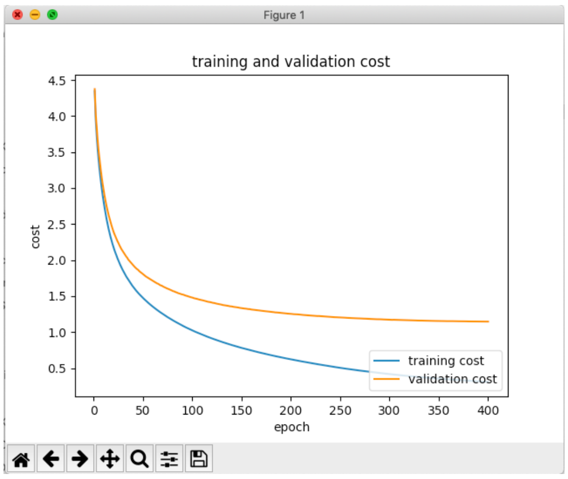


Fig: learning rate = 0.05, batch = 30

Now, if we analyse this problem scenario by means of those graphs and outputs we have following statistical table:

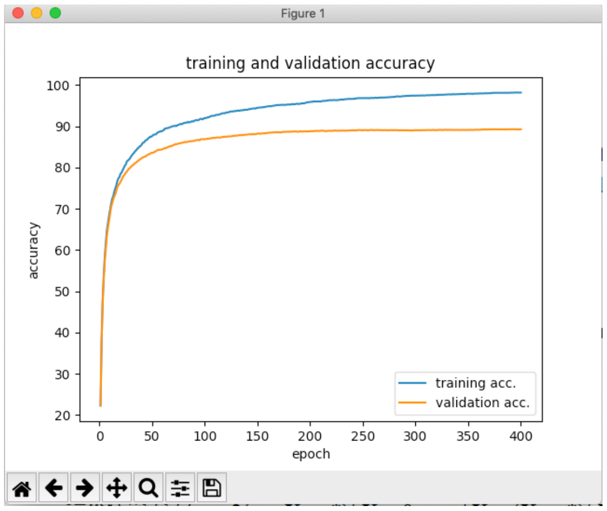
Conf Number	Learning Rate	Number of Batch	Training Cost	Validation Cost	Training Accuracy	Validation Accuracy	Test Accuracy
1	0.001	10	0.6173	1.7953	64.37	55.41	54.630000
2	0.005	10	0.5427	1.3451	87.5240	73.0245	74.350000
3	0.010	10	0.00253	1.7821	97.0241	78.5624	78.350000
4	0.05	10	0.009867	1.5	100	79.0658	80.620000
5	0.1	10	0.008361	1.6015	100	82.9967	81.880000
6	0.5	10	0.0029	1.6	93.23	83.07	82.300000
7	0.05	1	0.009837	1.6729	98.5284	81.2591	82.470000
8	0.05	5	0.03625	1.5981	99.2176	80.2267	81.620000
9	0.05	15	0.05138	1.5273	100	78.2088	80.600000
10	0.05	30	0.02736	1.4029	99.5543	79.6397	80.670000

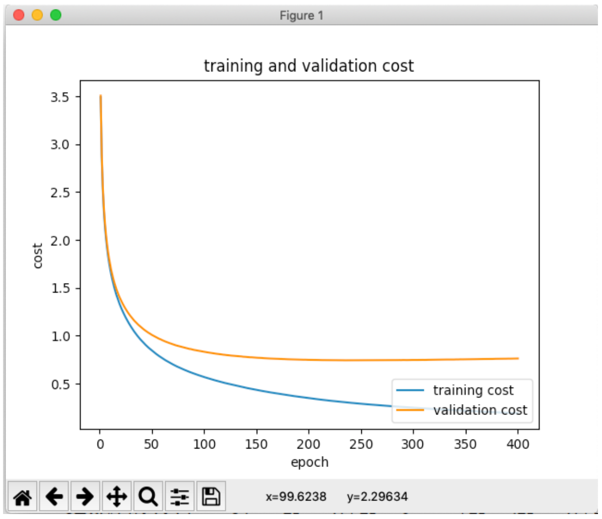
Findings: So far, we have checked different configuration changing learning rate and number of batch. From this statistics we are sure that the overfitting effect doesn't depends on learning rate and number of batch.

Reduce effect of Overfitting:
There are several ways to reduce Overfitting effect. Among them we can choose two approach to have our desired test accuracy.

i) Adding more Training Sample
ii) Reduce some parameter.

i) Adding more training Sample: In our previous analysis we use 1000 samples. Now we have increase the number of samples by 4500. After adding more training samples we have come up with following graph:

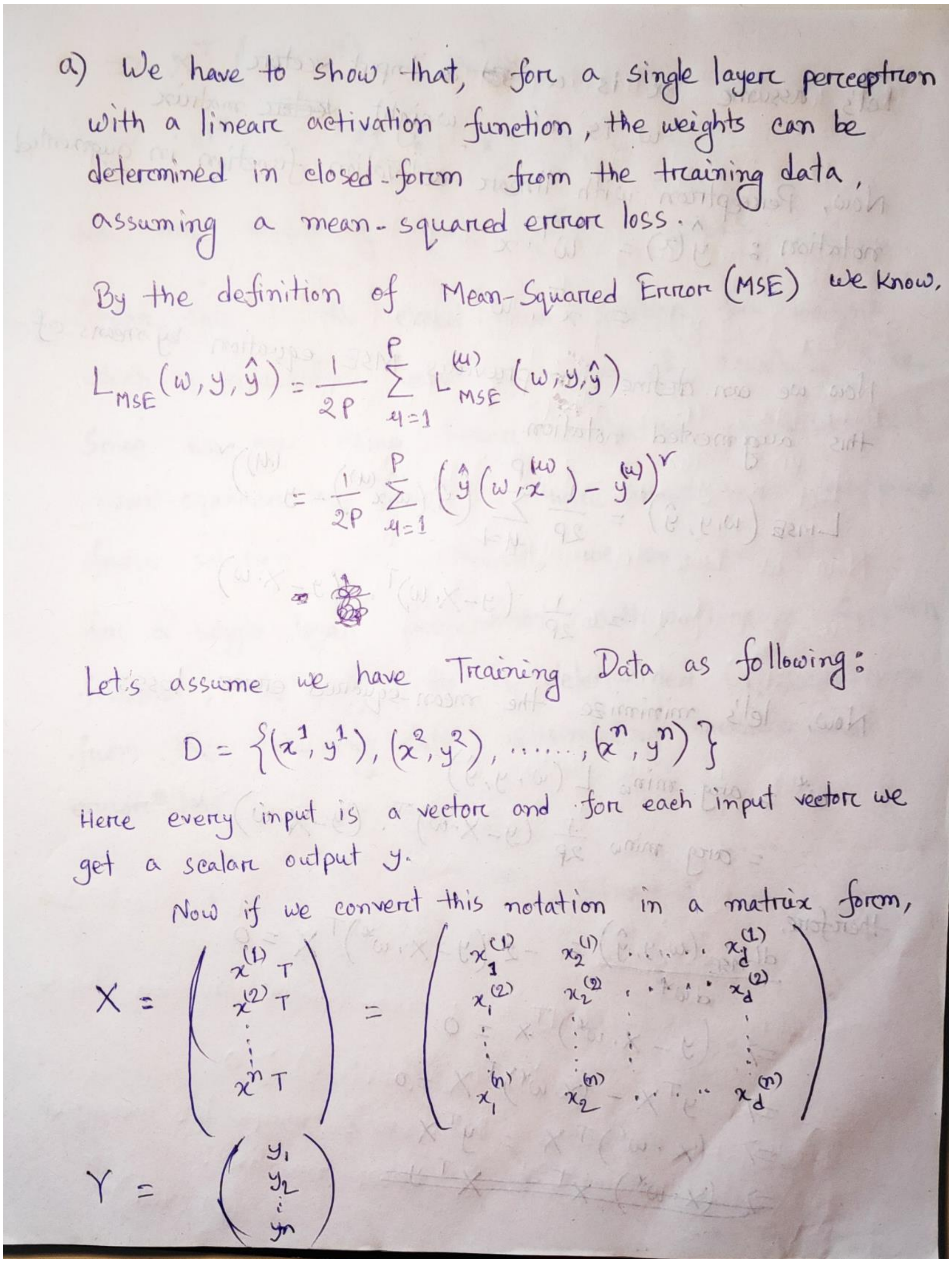




Findings: After adding only 3500 more samples with learning rate 0.05 we got the test accuracy 89.62% which is way better than our previous analysis. So, it is clear to us that if we add more samples for this problem scenario, it will prevent overfitting problem.

ii) Reduce some parameter: In our analysis we used one hidden layer with 30 hidden neurons. There might be a possibility if we test this problem scenario without any hidden layers. But unfortunately we didn't get any better test accuracy by reducing parameters.

Exercise 2 (Loss functions and weight update formulae, theoretical considerations):



a) We have to show that, for a single layer perceptron with a linear activation function, the weights can be determined in closed-form from the training data, assuming a mean-squared error loss.

By the definition of Mean-Squared Error (MSE) we know,

$$L_{MSE}(w, y, \hat{y}) = \frac{1}{2P} \sum_{\mu=1}^{P} L_{MSE}^{(\mu)}(w, y, \hat{y})$$

$$= \frac{1}{2P} \sum_{\mu=1}^{P} \left(\hat{y}(w, x^{(\mu)}) - y^{(\mu)} \right)^2$$

$=$ [illegible]

Let's assume we have Training Data as following:

$$D = \{(x^1, y^1), (x^2, y^2), \ldots\ldots, (x^n, y^n)\}$$

Here every input is a vector and for each input vector we get a scalar output y.

Now if we convert this notation in a matrix form,

$$X = \begin{pmatrix} x^{(1)\,T} \\ x^{(2)\,T} \\ \vdots \\ x^{n\,T} \end{pmatrix} = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_d^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_d^{(2)} \\ \vdots & \vdots & & \vdots \\ x_1^{(n)} & x_2^{(n)} & \cdots & x_d^{(n)} \end{pmatrix}$$

$$Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Let's assume x is a ~~set~~ ~~of~~ input vector

w is a weight ~~vector~~ matrix

Now, Perceptron with linear activation function in augmented notation : $\hat{y}(x) = w^T \cdot x$

$$= x^T \cdot w$$

Now we can define our previous MSE equation by means of this augmented notation.

$$L_{MSE}(w, y, \hat{y}) = \frac{1}{2P} \sum_{\mu=1}^{P} \left(\hat{y}(w, x^{(\mu)}) - y^{(\mu)} \right)^2$$

$$= \frac{1}{2P} (y - X \cdot w)^T \cdot (y - X \cdot w)$$

Now, let's minimize the mean-squared error loss:

$$w^* = \arg\min_w L(w, y, \hat{y})$$

$$= \arg\min_w \frac{1}{2P} (y - X \cdot w)^T \cdot (y - X \cdot w)$$

therefore,

$$\frac{dL_{MSE}(w, y, \hat{y})}{dw^*} = -2 (y - X \cdot w^*)^T X = 0$$

$$\Rightarrow (y - X \cdot w^*)^T X = 0$$

$$\Rightarrow y^T X - (X \cdot w^*)^T X = 0$$

$$\Rightarrow (X \cdot w^*)^T X = y^T X$$

~~$\Rightarrow (X \cdot w^*) X^T = X^T y$~~

$$\Rightarrow X^T (X \cdot w^*) = X^T y$$

$$\Rightarrow w^* = \cancel{\varnothing} (X^T X)^{-1} X^T y$$

we are assuming that $(X^T X)^{-1}$ exists.

So far, this is the closed form solution for weights which is calculated ~~using~~ by means of training data. Since we are using linear activation function and mean-squared loss function, there must exists a closed form solution. As a result, we can say that, for a single layer perceptron with a linear activation function, the weights can be determined in closed-form from the training data, assumming mean-squared error loss.

(Showed)

(b) We have a training set

$$D = \{(x^1, y^1), (x^2, y^2), \ldots, (x^n, y^n)\}$$

By definition of cross-entropy loss we have:

$$L_{CE}(w, y, \hat{y}) = -\frac{1}{P} \sum_{\mu=1}^{P} \left[y^{(\mu)} \ln \hat{y}(w, x^{(\mu)}) + (1-y^{(\mu)}) \ln (1-\hat{y}(w, x^{(\mu)})) \right]$$

therefore, cross-entropy loss for each sample,

$$L_{CE}^{(\mu)} = -y^{(\mu)} \ln \hat{y} - (1-y^{(\mu)}) \ln (1-\hat{y})$$

Now we have to analyse the "ln" function.

For $x \in [0,1]$ we know that $\ln(x) \leq 0$

also, $\lim_{x \to \infty} x \ln(x) = 0$

So, we are sure that,

$$y^{(\mu)} \ln \hat{y} \leq 0 \quad \text{for all } y, \hat{y} \in [0,1]$$

Similarly we can also assume that,

$$(1-y^{(\mu)}) \ln (1-\hat{y}) \leq 0 \quad \text{for all } y, \hat{y} \in [0,1].$$

Therefore, cross-entropy loss is non-negative for each sample.

$$L_{CE}^{(\mu)} = -y^{(\mu)} \ln \hat{y} - (1-y^{(\mu)}) \ln (1-\hat{y}) \geq 0$$

Finally, we can also say that, cross-entropy loss averaged over whole training set is non-negative.

(proved)

Now we have to calculate the minimum of the cross-entropy loss and show that it corresponds to the target value $y(\mu)$ for each training sample μ.

Applying derivation ~~we~~ on cross-entropy loss for each sample we get,

$$\frac{d L_{CE}^{(\mu)}}{d \hat{y}} = - \frac{y^{(\mu)}}{\hat{y}} + \frac{1-y^{(\mu)}}{1-\hat{y}} = 0$$

$$\Rightarrow \frac{y^{(\mu)}}{\hat{y}} = \frac{1-y^{(\mu)}}{1-\hat{y}}$$

$$\Rightarrow y^{(\mu)}(1-\hat{y}) = \hat{y}(1-y^{(\mu)})$$

$$\Rightarrow y^{(\mu)} - y^{\mu}.\hat{y} = \hat{y} - \hat{y}.y^{(\mu)}$$

$$\Rightarrow y^{(\mu)} = \hat{y}.$$

Now, it is clear to us that, for a single sample cross-entropy loss is minimized if the perceptron output $\hat{y}$ corresponds to the target $y^{(\mu)}$.

(Showed)

Exercise 3 (Convolutional neural networks):

a) Explaining following terms:

- **Convolutional Neural network**: A convolutional neural network is a class of deep neural network, most commonly applied to analyzing visual imagery. In short form it is represented as CNN or ConvNet. CNNs are regularized versions of multilayer perceptrons. Multilayer perceptrons usually refer to fully connected networks, that is each neuron in one layer is connected to all neurons in the next layer. The "fully-connectedness" of these networks make them prone to overfitting data. Typical ways of regularization includes adding some form of magnitude measurement of weights to the loss function. However, CNNs take a different approach towards regularization: they take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns. Therefore, on the scale of connectedness and complexity, CNNs are on the lower extreme.
  When programming a CNN, each convolutional layer within a neural network should have the following attributes:
  - Input is a tensor with shape (number of images) x (image width) x (image height) x (image depth).
  - Convolutional kernels whose width and height are hyper-parameters, and whose depth must be equal to that of the image. Convolutional layers convolve the input and pass its result to the next layer. This is similar to the response of a neuron in the visual cortex to a specific stimulus.
- **Filter, Kernel**: A filter (or kernel) is an integral component of the layered architecture. Generally, it refers to an operator applied to the entirety of the image such that it transforms the information encoded in the pixels. In practice, however, a kernel is a smaller-sized matrix in comparison to the input dimensions of the image, that consists of real valued entries. The kernels are then convolved with the input volume to obtain so-called 'activation maps'. Activation maps indicate 'activated' regions, regions where features specific to the kernel have been detected in the input. The real values of the kernel matrix change with each learning iteration over the training set, indicating that the network is learning to identify which regions are of significance for extracting features from the data.
- **Feature Map:** A feature map is the result of applying a particular kernel to an input. The size of the input image, the size of the filter and the type of padding determine the size of the feature map. Each feature map represents the presence of a particular feature in the input image at various locations in the image. Detecting several features in the input image corresponds to applying several filters to the input image. Each filter leads to its own feature maps.
- **Receptive field:** When dealing with high dimensional inputs such as images, it is impractical to connect neurons to all neurons in the previous volume. Instead, we connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a hyper-parameter called the **Receptive field** of the neuron.
- **Pooling:** A pooling or subsampling layer is a layer in a convolutional neural network which combines the activations of several units into a single unit of the current layer. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network. Pooling layer operates on each feature map independently. The most common approach used in pooling is max pooling which simply uses the maximum activation of all units in the receptive field as new

values. Average pooling uses the average value of the unit's activations within the receptive filed.

- **Fully convolutional network**: A fully convolutional network is network without any fully connected layers. This means that all layers in the network are convolution layers. A CNN with fully connected layers is just as end to end learnable as a fully convolutional one. The main difference is that the fully convolutional net is learning filters everywhere. Even the decision making layers at the end of the network are filters. A fully convolutional net tries to learn representations and make decisions based on local spatial input. Appending a fully connected layer enables the network to learn something using global information where the spatial arrangement of the input falls away and need not apply.

b) CNN with keras, tensorflow with source code exercise3b.py:

So far we have installed following dependencies:

i) pydot
ii) graphviz
iii) keras
iv) tensorflow

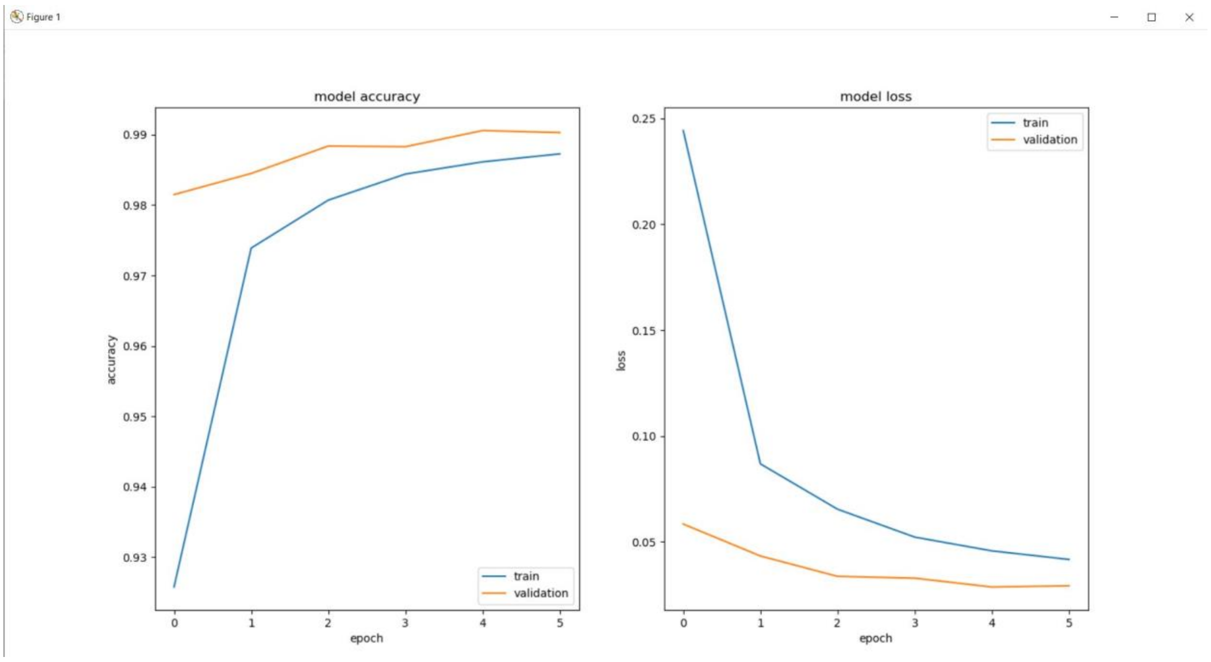
Now we have following implementation:

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Apart from this, we are setting following parameters:

Batch size : 128
Number of classes: 10
Epochs : 6 (We are using less epochs to get the output fast)
Optimizer : AdaDelta
Learning rate: 1.0

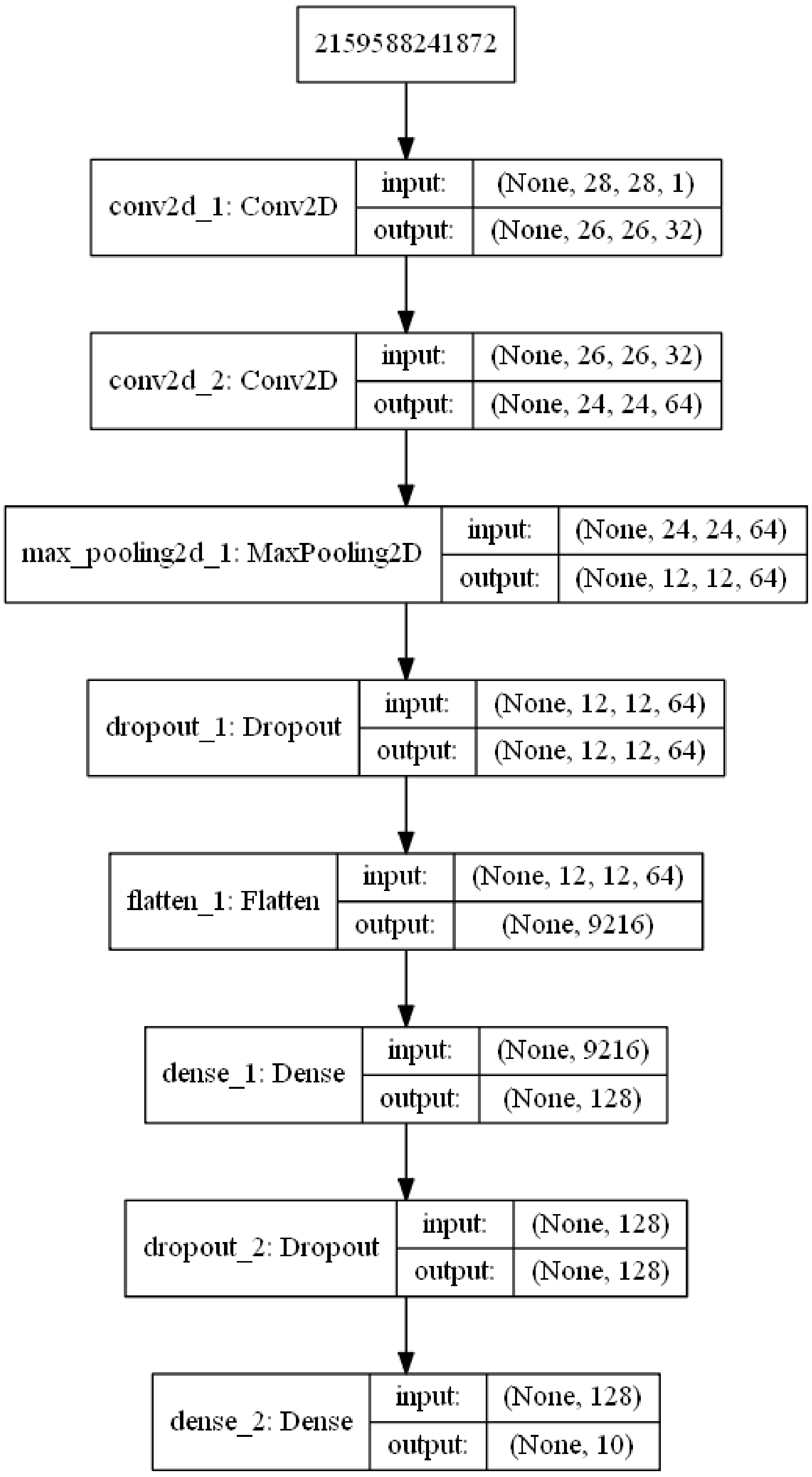
After running this script we got following graphs:



Here are our script output:

```
Epoch 2/6

60000/60000 [==============================] - 152s 3ms/step - loss: 0.0869 - acc: 0.9739 - val_loss: 0.0434 - val_acc: 0.9845

Epoch 3/6

60000/60000 [==============================] - 153s 3ms/step - loss: 0.0655 - acc: 0.9807 - val_loss: 0.0338 - val_acc: 0.9884

Epoch 4/6

60000/60000 [==============================] - 154s 3ms/step - loss: 0.0523 - acc: 0.9844 - val_loss: 0.0329 - val_acc: 0.9883

Epoch 5/6

60000/60000 [==============================] - 155s 3ms/step - loss: 0.0458 - acc: 0.9861 - val_loss: 0.0287 - val_acc: 0.9906

Epoch 6/6

60000/60000 [==============================] - 153s 3ms/step - loss: 0.0418 - acc: 0.9873 - val_loss: 0.0293 - val_acc: 0.9903

Number of parameters: 1199882

Test loss: 0.029315848156595894

Test accuracy: 0.9903
```

We also got following model structure:


2159588241872
conv2d_1: Conv2D
input:
(None, 28, 28, 1)
output:
(None, 26, 26, 32)
conv2d_2: Conv2D
input:
(None, 26, 26, 32)
output:
(None, 24, 24, 64)
max_pooling2d_1: MaxPooling2D
input:
(None, 24, 24, 64)
output:
(None, 12, 12, 64)
dropout_1: Dropout
input:
(None, 12, 12, 64)
output:
(None, 12, 12, 64)
flatten_1: Flatten
input:
(None, 12, 12, 64)
output:
(None, 9216)
dense_1: Dense
input:
(None, 9216)
output:
(None, 128)
dropout_2: Dropout
input:
(None, 128)
output:
(None, 128)
dense_2: Dense
input:
(None, 128)
output:
(None, 10)