TUTORIAL DOCUMENTATION

DEEP LEARNING TUTORIALS

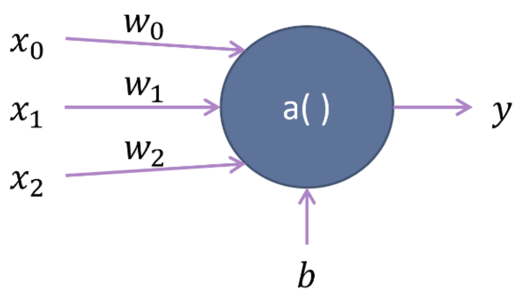
LEON HOFFMANN

Inhalt

Basics

Mathematical model of a neural network

In this figure you can see the model of a neuron. The inputs x_i on the left are some inputs based on the dataset ore some outputs from other neurons. These inputs getting “weighted” by the weights w_i. The neuron itself contains a function which calculates the output y. There is also a bias b. The bias helps to shift the activation function into the right direction, this helps for the learning success.



This sketch of a neuron can be described by a mathematical formula. The inputs and weights are some matrices. There are i inputs ore outputs from the neurons of the layer before and n neurons in the described layer.

$$\underline{w} \cdot \underline{x} = \underline{y}$$

$$\begin{bmatrix} w_{0,0} & \cdots & w_{0,i} \\ \vdots & \ddots & \vdots \\ w_{n,0} & \cdots & w_{n,i} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \cdots \\ x_i \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \cdots \\ b_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \cdots \\ y_n \end{bmatrix}$$

The neuron itself can be describes by the following formula. It adds up all multiplications of the inputs and the weights. After that the value is adjusted by an activation function $a(\,)$. This can be the sigmoid or another function which flattens extreme outputs. For the sigmoid y is between 0 and 1.

$$y_n = a\left(\sum_i w_{n,i}\, x_{n,i} + b_n\right) = a\left(\sum_i s(w, x, b)\right)$$

1. Backpropagation and introduction to TensorFlow

1.1 Backpropagation

1.1.1 The loss function

The backpropagation is a method to train a neuronal network. The idea is to adjust the weights for every neuron based on the value it puts out $\hat{y}$ and the expected value $\tilde{y}$. To calculate the adjustment of the weights a loss function is used.

$$L = \frac{1}{n}\sum_{k}^{n}(\tilde{y}_k - \hat{y}_k)^2$$

The following formula shows the architecture of an example neuron and the code the implementation in Python using the `numpy` library

$$a(\underline{x}, \underline{w}) = \left|\left|\underline{w} \cdot \underline{x}\right|\right|^2 = \left|\left|\underline{q}\right|\right|^2$$

The implementation is split into two functions, the first one dose the multiplication the second one is squaring the values and sums them up to get the predicted value $\tilde{y} =$ `y_pred`.

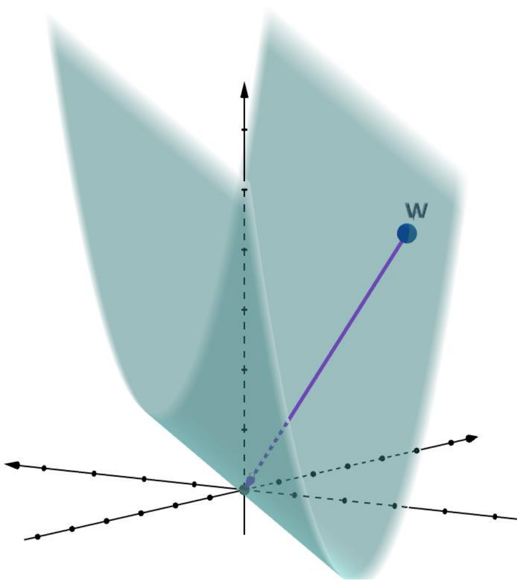
```
def multiplication (W, x):
    q = np.dot (W, x)
    return q

def prediction(q):
    f_1 = q*q
    y_pred = np.sum(f_1)
    return y_pred
```

This code shows the final calculation of the loss. The first line is subtracting the expected value $\hat{y} =$ `y` from the predicted value $\tilde{y} =$ `y_pred`. This is called the error. The function is returning the loss which is the error squared.

```
def prediction_loss(y_pred, y):
    error = y_pred - y
    loss = np.square(error)
    return loss
```

1.1.2 Weight adjustment and gradient calculation

The neuronal network is learning by adjusting the weights. Basically, the system tries to make the error as small as possible, by finding the low point of a function. The following picture is showing an example of a function with two inputs. The lowest points in each direction are the perfect values for each weight, in this example the origin. The got the point w (in blue) as weights. It should move to the lowest point, the perfect value. By calculating the gradient for each direction, the point can be moved stepwise into the right direction.



To calculate the gradients the partial derivates need to be implemented. The following formula shows how to calculate the gradients based of the partial derivate.

$$\frac{\partial L}{\partial w} = \frac{\partial q}{\partial w} \cdot \frac{\partial y}{\partial q} \cdot \frac{\partial L}{\partial y}$$

$$\frac{\partial L(\underline{x}, \underline{w})}{\partial \underline{w}} = \frac{\partial L(q)}{\partial q} \cdot \frac{\partial q}{\partial \underline{w}} = 2 \cdot q \cdot \underline{x}^T$$

The following code corresponds to the gradient of the loss. It is the partial derivate of the loss function.

$$\frac{\partial L}{\partial y} = 2 \cdot (\tilde{y} - \hat{y})$$

```
def gradient_loss(y_pred, y):
    grad_loss = 2*(y_pred-y)
    return grad_loss
```

The next step is the gradient of the prediction function. This is following the shown formula and code.

$$\frac{\partial L}{\partial q} = \frac{\partial y}{\partial q} \cdot \frac{\partial L}{\partial y} = 2 \cdot q \cdot 2 \cdot (\tilde{y} - \hat{y})$$

```
def gradient_prediction (q, grad_loss):
    grad_q = 2*q*grad_loss
    return grad_q
```

And finally, the gradients of all parameters. If the dimension of $\underline{x}$ is greater than 1 a scalar multiplication is needed. This function returns an array with all parameter gradiants.

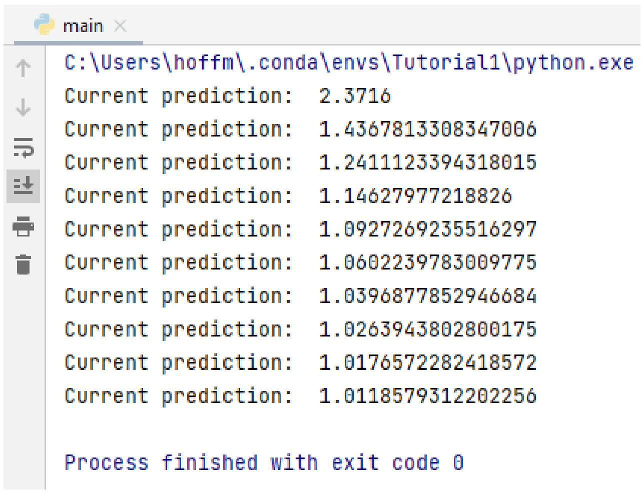
$$\frac{\partial L}{\partial w} = \frac{\partial q}{\partial w} \cdot \frac{\partial L}{\partial q} = \underline{x} \cdot 2 \cdot q \cdot 2 \cdot (\tilde{y} - \hat{y})$$

```
def gradient_multiplication (x,
grad_q):
    if len(x) == 1:
        grad_W = grad_q * x.T
    else:
        grad_W = grad_q.dot(x.T)
    return grad_W
```

The final step to get the neurons to learn is to update the values in the weight matrix. This is done by subtracting the gradients of each parameter. So, the values are moving downwards into the right direction, the lowest point. To prevent overstepping the gradient is multiplied by a typically small value <1, the learning rate.

```
def update (W, grad_W, learning_rate):
    W = W - learning_rate*grad_W
    return W
```

By iterating over and over the values with der `update` function the network is getting better and better. In the following terminal screen is showing how the prediction is getting better ($\hat{y} = 1$). The table shows the initial values and the values after 10 iterations.



```
main
C:\Users\hoffm\.conda\envs\Tutorial1\python.exe
Current prediction:  2.3716
Current prediction:  1.4367813308347006
Current prediction:  1.2411123394318015
Current prediction:  1.14627977218826
Current prediction:  1.0927269235516297
Current prediction:  1.0602239783009775
Current prediction:  1.0396877852946684
Current prediction:  1.0263943802800175
Current prediction:  1.0176572282418572
Current prediction:  1.0118579312202256

Process finished with exit code 0
```

iterations	w	Y
i = 0	[-0,3 0,8]	-
i = 1	[-0.316 0.631]	2,3716
i = 10	[-0.326 0.534]	1,01185

1.2 Optimization

Code review and explanation

1.2.1 Define graph / network

This section creates the initial weights and biases of the network in the form of a matrixes. There are three weights in the weight's matrix and one bias value in the bias matrix.

1.2.2 Unit design

For the unit design, a new class which inheritances a TensorFlow Keras model is created. This will be a neuron. It contains the weights and biases matrixes. For an easy initialization the shown function is used.

```
def __init__(self, weights, biases)
```

Further a function named call is implemented. This function defines the structure of the neuron and returns a prediction based on the inputs.

```
__call__(self, x0)
```

This is done by multiplying the input x matrix and the weight w matrix. After that the biases are getting added. This calculation defines the layer. To get the prediction, the "relu" function is used. This is defined as following, so the prediction is zero or positive.

$$y_{relu} = \begin{cases} x: x > 0 \\ 0: x < 0 \end{cases}$$

1.2.3 Let the model learn

The weights and biases are saved in matrixes. The following function is watching the executed operations and calculates this way automatically the gradients.

```
tf.GradientTape()
```

Inside this function the input matrix x is handed over to the network. The output is the predicted value. After that the loss function is calculated based on the mean square error and the parameters predicted and expected value. After that the gradients are calculated as shown.

```
gradients = tape.gradient(loss_value, y_pred.trainable_weights)
```

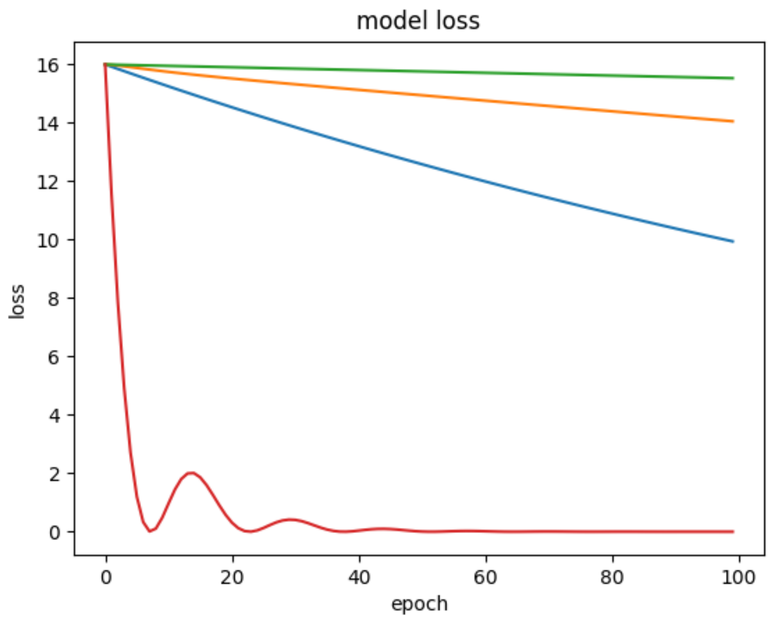
The final step is to adjust the weights. This is done by the following line.

```
opt.apply_gradients(zip(gradients, y_pred.trainable_weights))
```

If this process is repeated several times the network is learning. The gradients are applied as a function of `opt`. This is an optimizer, which optimizes the learning process.

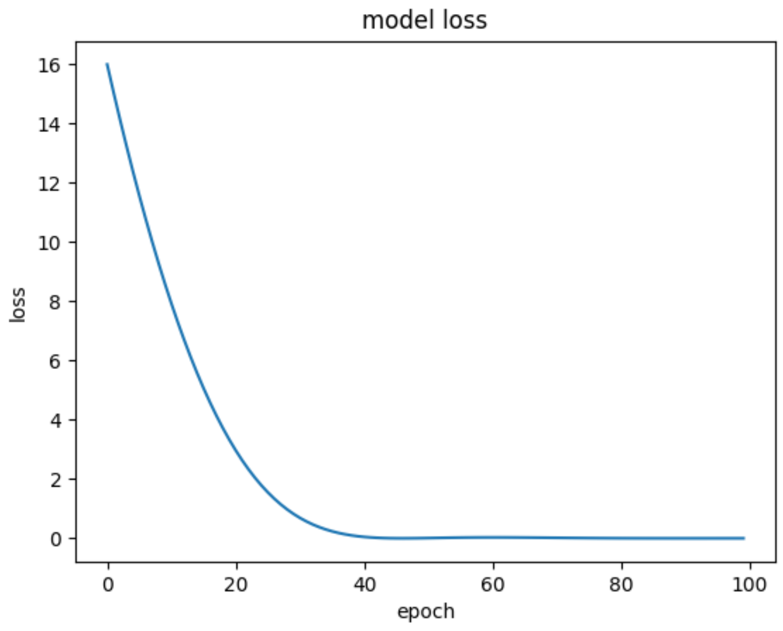
1.2.4 Compare some optimizers

This graph is showing some optimizers and their impact to the loss in comparison. The green and red graph are from the same optimizer called adam, just with different learning rates.



🟦 Stochastic gradient descent
🟨 Momentum optimizer
🟩 Adaptive momentum optimizer learning rate = 0.0001
🟥 Adaptive momentum optimizer with adjusted initial learning rate = 0.1

With a good learning rate and the adam optimizer the graph can look like this.



1.3 Regularization

To understand regularization the Keras library and the MNIST data set are used in this chapter.

1.3.1 Define a model with Keras

The regularization begins with the model design. For comparability the models which are compared differ only slightly. The basic model includes a Flatten layer (28 x 28) as the input layer. Followed by five dense layers with 512 units each and the activation function Relu.

```
tf.keras.layers.Dense(512, activation=tf.nn.relu)
```

The last layer, the output, is a dense layer with 10 units and the activation function SoftMax. The base design is done, the next step is to compare the impact of regularization.

1.3.1.1 Model learning with Keras

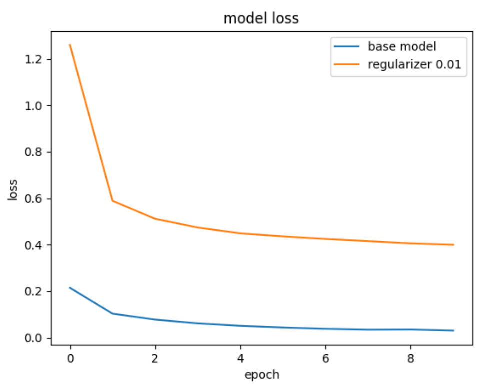
Each model is going to learn with the same parameters. The following code is showing how

1.3.2 Compare regularization to the base model

This table is showing the impact of different regularizations.

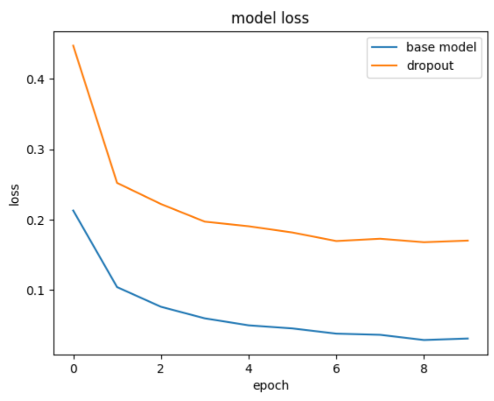
Base model reference	loss: 0.0312 - accuracy: 0.9918

L2 regularization term with $\lambda = 0.01$



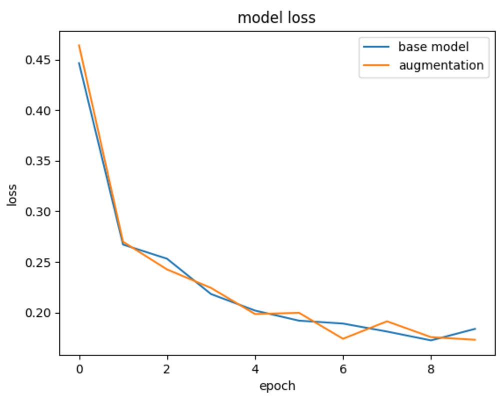
loss: 0.7688 - accuracy: 0.8540

Dropout 0.5 every layer



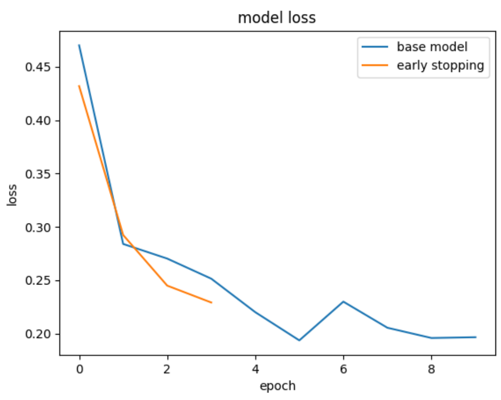
loss: 0.1704 - accuracy: 0.9594

Augmentation



loss: 0.1730 - accuracy: 0.9599

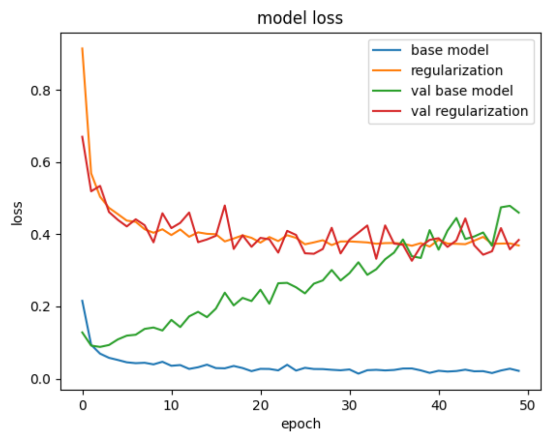
Early stopping



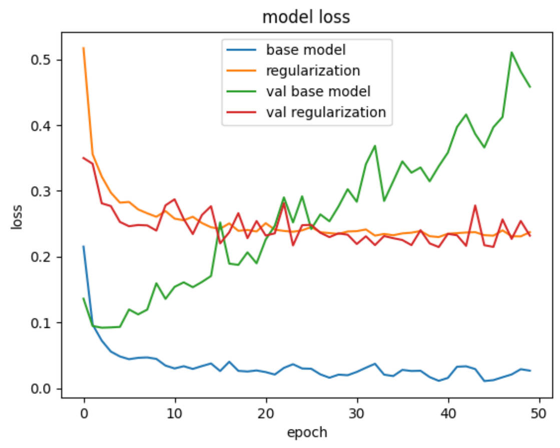
loss: 0.1965 - accuracy: 0.9584

These results are sobering, cause the loss with a regularization is higher then without. But these examples are very simple, so it doesn't need much to get a good loss. Now the comparison of some harder examples. This time the loss of the learning and the validation loss are plotted. Epochs are set to 50 and the learning rate is increased from 0,001 to 0,01. You can see how the validation loss of the run without a regularization is drifting up, while the loss is relatively flat. This is called overfitting. The System can recognize already learned data very good, but the validation set increasingly bad. What also stands out is, the lower the regularization factor is, the lower is the overall loss. Compare 0.01 with a loss around 0.4 and the regularization 0.001 with a loss about 0.25. If the regularization factor is too low, the effect of preventing overfitting is getting lost. Example regularization 0.0001

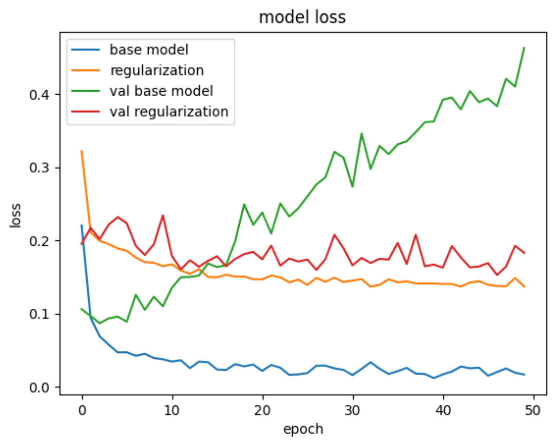
Regularization = 0.01



Regularization = 0.001



Regularization = 0.0001



2. Transfer Learning with TensorFlow for Object Classification

This chapter is about pre-trained models. The example dataset consists of images from cats and dogs. This dataset got 1000 pictures of dogs and 1000 pictures of cats.

2.1 Prepare the data

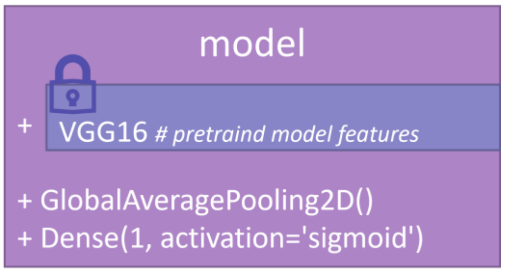
The pictures are stored in a folder structure. The first step is to write their path into variables so they can be used in the code. After that, the pictures are getting resized to the same size. That way the network only needs to handle one kind of picture. This is done by a `ImageDataGenerator`. It puts the images into batches and resizes them. This resize-operation is another kind of augmentation. So, the training and validation data is ready.

2.2 Prepare pretrained model

The pretrained model, which is used is called the VGG16 model. It can be used for all kinds of image recognition. For using it some adjustments need to be done. First the input layer needs to be set to (200 x 200 x 3), the image size and the RGB colour properties. The weights are already pretrained. This way the network got already an idea how to recognize pictures. With these arguments the needed model can be downloaded.

2.3 Feature extraction

The downloaded pretrained model contains already trained useful layers. These features can be included into another model. The following picture is showing how the VGG16 is included into an example model. The pretrained layers are set to not trainable, so only the two added layers are gone be adjusted. This was the learning process is much faster, cause just some matrixes need to be calculated. With the pretrained model and the two extra self-trained layers, a network with good results can be implemented quickly.


model
+ VGG16 # pretraind model features
+ GlobalAveragePooling2D()
+ Dense(1, activation='sigmoid')

2.3.1 Compare full and limited training

This table is showing the time saving based caused by not fully training the model, but just the two extra layers. The time is nearly quartered.

Batch size	Training duration / epoch	
	Limited training	Full training
16	ETA: 11:23	ETA: 35:24
32	ETA: 9:44	ETA: 32:35
64	ETA: 9:02	ETA: 31:12
128	ETA: 8:14	ETA: 28:14

3. Semantic segmentation with U-Net

3.1 Semantic segmentation

Sematic segmentation assigns to every pixel a category, so the network can tell where an object is located in a picture. This case where the cell and where the cell border is.

3.2 U-Net

The following model is using a u-net. This net is first chopping the input down using convolutional layers and scales them up again.

3.3 Data generation

There are only 30 pictures in the dataset. This is not enough to train a network. The given pictures are used in different ways to expand the dataset. They get rotated, flipped and zoomed in into different areas. Important is, that the label data is manipulated in the same way.

3.4 Model layers

3.4.1 Convolutional layer

The convolutional layer can recognize patterns. This is done by chopping the pixel level down with a filter, so a specific composition of pixels, results in a certain type of new structure. In this way circles, lines or curves can be specifically recognized. This layer property is practical for image recognition because the network learns not only on the pixel level, but on a higher structure layer.

3.4.2 MaxPooling

This layer pools the pixels together, so the strongest features are getting summarized in a new sized matrix. This layer works especially with the convolutional layer very good together. Because the first layer is putting out some structures and the second layer puts out the most prominent points.

Literature

https://www.tensorflow.org/

https://keras.io/

https://www.youtube.com/watch?v=aircAruvnKk&t=1007s

https://medium.com/analytics-vidhya/image-augmentation-using-keras-99072b490c72