University of Constance
Department of Computer Science & Information Science
Jonas Probst, Simon Giebenhain, Gabriel Scheibler, Clemens Gutknecht

Solution of 4. Assignment

Deadline: 28.1.18

Points: (maximum 110 Points)

Exercise 1	Exercise 2	Exercise 3	Exercise 4	**Total**
20	60	10	20	110

Exercise 1: Transposed Convolutions

a)

One possible solution is:
$x1 = 17, x2 = 15$
$x3 = 3, x4 = 2$
$x5 = 15, x6 = 13$
$x7 = 2, x8 = 6$

b)

```
W3 = tf.Variable(tf.zeros([3,3,15,13]))
layer3 = tf.nn.conv2d_transpose(layer2, W3, output_shape=tf.constant(np.array([-1,22,24,15],
dtype=np.int32)), strides=[1,2,6,1], padding = 'SAME')
W4 = tf.Variable(tf.zeros([3,3,17,15]))
layer4 = tf.nn.conv2d_transpose(layer3, W4, output_shape=tf.constant(np.array([-1,64,48,17],
dtype=np.int32)), strides=[1,3,2,1], padding = 'SAME')
layer4.shape
```

c)

It's not clear how much padding was used in the convolution, so output shape needs to be specified in the method.

[ ] of 20 Points

Exercise 2: Encoder-Decoder

a)

The method **normalize_mean()** and **normalize_stddev()** normalize the mean and standard deviation of the input data respectively.

b)

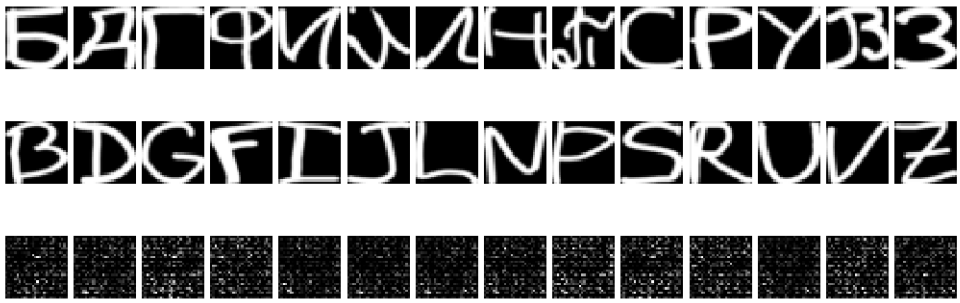
The architecture of our encoder-decoder is described in the following:
Encoder:
The inputs are images of size $[28, 28, 1]$.
We apply a strided convolution with kernel dimensions $[3, 3, 1, 32]$ and strides $[1, 2, 2, 1]$ followed by basically the same convolution with kernel dimensions $[3, 3, 32, 64]$ and strides $[1, 2, 2, 1]$. This yields an output of the following shape $[7, 7, 64]$.
The last step of the encoder is a fully connected layer, which reduces the data to a vector of size 512.
A dropout layer connects encoder and decoder.

Decoder:
First the data of shape [batch_size, 512] is transformed to shape [batch_size, 7, 7, 64] with a fully connected layer and reshaping.
Afterwards we apply transposed convolutions mirroring the convolutions of the encoder, such that we come form shape $[7, 7, 64]$ to $[28, 28, 1]$ per image. An additional fully connected layer, which maintains shape, follows the convolutions.

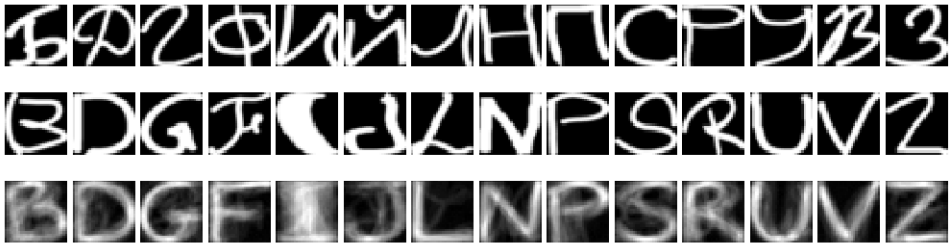
Further Details:
We used *parametric ReLU* as activation function and applied *batch normalization* before every activation.
As mentioned above we applied a *droupout-layer* inbetween encoder and decoder.

c)

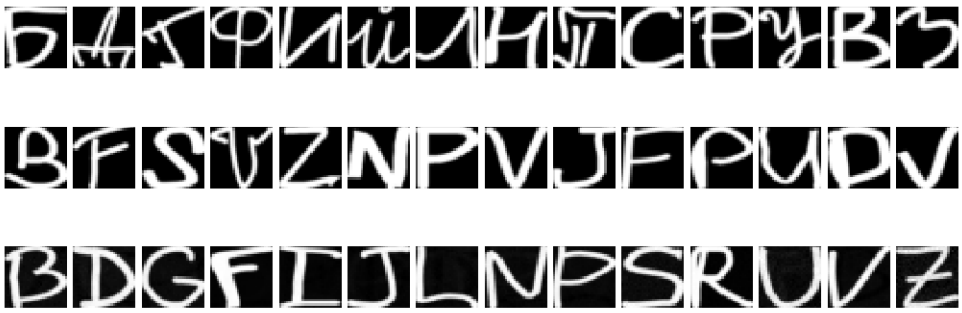
We struggled to come up with a sensible training method. At first we believed, that classifying(pretrained on the latin training dataset) the output of the encoder-decoder and apllying cross entropy to the result, would be the best idea. We followed this approach in **encoder_decoder_class.py**. We achieved quit solid results, with a classification accuracy of around 97%. However, the outputs of the encoder-decoder were not readable for humans. This is due to the fact, that the classifier classifies every input he gets, and cannot distinguish between a nice letter and some extremly noisy letter. We tried to fix this, by introducing a noise class, with randomly generated training examples labeled as noise, but it did not help.
Here is a visualization of the results:



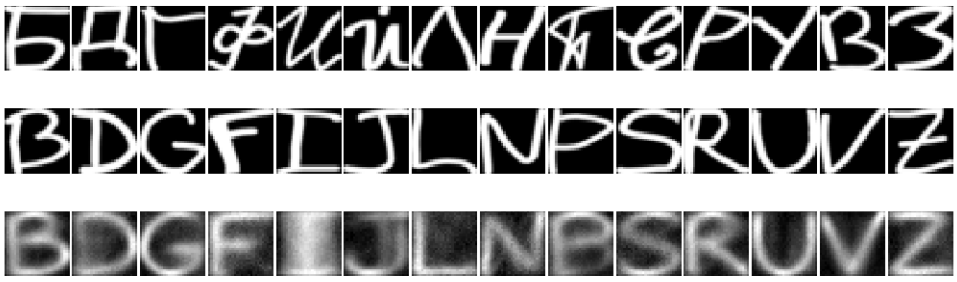
Another approach for the loss function, is to take the L2-loss of the output of the encoder-decoder with random examples of latin letters of the same class. This approach seemed to work quite well. However the outputs are always very similar within each class.



One could also choose a representative for each class of the latin latters and always compute the L2-loss of the difference of the output of the encoder-decoder and the respective representative. This basically achieves, that for every cyrillic letter of class k the encoder-decoder outputs the representativ of class k. (The same could be achieved with simple classification and return the representative of that class)
Here is a visualization:



Also one could always compare with the mean of the respective class, which does not really make sense, because the network is then trained to basically classify the image and return the mean of the resulting class. Here are the results:



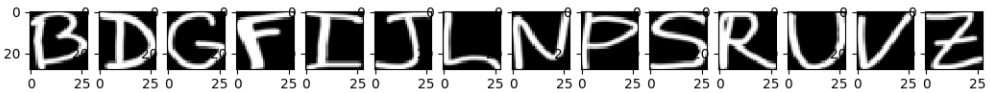
Tricks: Mentioned above in (b)

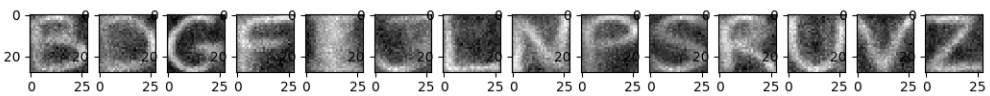
d)

See (c).

e)

For this exercise we used the training strategy and weights of **encoder_decoder_random.py**. The script **latent_space.py** loades the weights saved in **encoder_decoder_random.train_for_latent_space()**. The encoder part of the network is obmitted, it begins with the 512-tupel, which was in the middle before. All variables are markes as untrainable and only the 512-tupel is trainable. We optimezed for every class, the loss for a single class is the MSE of the output of the encoder and all training examples of that class. The results are shown below:





[ ] of 60 Points

Exercise 3: Backpropagation trough a ConvLayer

a)

We can interpret A as a function $F_A : \mathbb{R}^n \to \mathbb{R}^m, x \mapsto Ax$ Applying the multidimensional chain rule

on $E(x) = g(Ax) = g(F_A(x))$ yields:

$$D_x E(x) = D_x g(F_A(x)) \cdot D_x F_A(x)$$

With the fact that $D_x E(x) = \nabla E(x)^T$, we get:

$$\nabla E(x) = (D_x g(F_A(x)) \cdot D_x F_A(x))^T = (\nabla g(F_A(x))^T \cdot A)^T = A^T \cdot \nabla g(F_A(x))$$

b)

This can be nicely explained with (a). We could interpret E as our loss, g as our activation function and A as a convolution (in the lecture we discussed that every convolution can be expressed with a matrix). If we want to apply some form of gradient descent, we have to compute the gradient of E, which is done using backpropagation. As we saw in (a) the gradient of E depends on ∇g and A^T. And from the lecture we know, that A^T is the transposed convolution to A (this is where the name comes from).

[ ] of 10 Points

Exercise 4: PCA vs. Auto-Encoder

a)

Let $x \in \mathbb{R}^d$. Let$v_1, \ldots, v_d$ be the d principal components, and $\tilde{V} := (v_1, \ldots, v_k)$ be the matric with the first k principal components.
The projection is done, in the following way: First the length of the projection of x onto each v_i is calculated by the dot product: $v_i^T x$, then v_i is scaled by this factor. Then all resulting vectors are combined to represent x in the k dimensional subspace. In compact form:

$$\sum_{i=1}^{k} (v_i^T x) v_i = (\tilde{V}\tilde{V}^T x) \in \mathbb{R}^d$$

b)

For this we make the assumption, that the data has zero mean, that is: $\frac{1}{L}\sum_{l=1}^{L} x_l = 0$ (otherwise one would first have to shift the data to achieve zero mean.)

The loss for the auto-encoder is the following:

$$\sum_{l=1}^{L} \|x - BAx\|_2^2$$

In terms of the PCA $A = \tilde{V}^T$ and $B = \tilde{V}$. Before we simplify this expression, two remarks:

$$v_i^T v_i = 1 \qquad \text{(for all } i\text{)} \tag{1}$$

$$v_i^T v_j = 0 \qquad \text{(for all } i \neq j\text{)} \tag{2}$$

Both follow, because the principal components are a orthonormal basis.

Furthermore x can be represented exactly, if all principal components are used(because they form a orthonormal basis), that is:

$$x = \sum_{i_1}^{d} (v_i^T x) v_i \tag{3}$$

and note the definition of the covariance matrix Σ

$$\Sigma := \frac{1}{L}\sum_{i=1}^{L} (x_i - \overline{x})(x_i - \overline{x})^L \overset{zeromean}{=} \frac{1}{L}\sum_{i=1}^{L} x_i x^L \tag{4}$$

There we can express the l2-loss in the following way:

$$
\begin{aligned}
\sum_{l=1}^{L} \|x - \tilde{V}\tilde{V}x_l\|_2^2 &\overset{(3)}{=} \sum_{l=1}^{L} \|(\sum_{i=1}^{d}(v_i^T x_l)v_i) - (\sum_{i=1}^{k}(v_i^T x_l)v_i)\|_2^2 && (5)\\
&= \sum_{l=1}^{L} \| \sum_{i=k+1}^{d} (v_i^T x_l)v_i\|_2^2 && (6)\\
&= \sum_{l=1}^{L} \left(\sum_{i=k+1}^{d} (v_i^T x_l)v_i) \right)^T \left(\sum_{i=k+1}^{d} (v_i^T x_l)v_i) \right) && (7)\\
&= \sum_{l=1}^{L} \left(\sum_{i=k+1}^{d} (v_i^T x_l)v_i^T \right) \left(\sum_{i=k+1}^{d} (v_i^T x_l)v_i) \right) && (8)\\
&= \sum_{l=1}^{L} \left(\sum_{i=k+1}^{d} \sum_{j=k+1}^{d} (v_i^T x_l)^2 v_i^T v_j) \right) && (9)\\
&= \sum_{l=1}^{L} \left(\sum_{i=k+1}^{d} (v_i^T x_l)^2 v_i^T v_i + \sum_{i=k+1}^{d} \sum_{i \neq j = k+1}^{d} (v_i^T x_l)^2 v_i^T v_j \right) && (10)\\
&\overset{(2)}{=} \sum_{l=1}^{L} \left(\sum_{i=k+1}^{d} (v_i^T x_l)^2 v_i^T v_i \right) && (11)\\
&\overset{(1)}{=} \sum_{l=1}^{L} \left(\sum_{i=k+1}^{d} (v_i^T x_l)^2 \right) && (12)\\
&= \sum_{i=k+1}^{d} \left(\sum_{l=1}^{L} (v_i^T x_l)(v_i^T x_l) \right) && (13)\\
&= \sum_{i=k+1}^{d} \left(\sum_{l=1}^{L} (v_i^T x_l)(x_l^T v_i) \right) && (14)\\
&\overset{(4)}{=} L \cdot \sum_{i=k+1}^{d} v_i^T \Sigma v_i && (15)\\
&= L \cdot \sum_{i=k+1}^{d} v_i^T \lambda_i v_i && (16)\\
&\overset{(1)}{=} L \cdot \sum_{i=k+1}^{d} \lambda_i && (17)\\
& && (18)
\end{aligned}
$$

Now we still have to prove, that $\tilde{V}$ is indeed a minimizer for $\sum_{l=1}^{L} \|x - WW^T x\|_2^2$.
For this we continue by minimizing (15) with respect to $v_{k+1}, \ldots, v_d$ under the condition, that $v_i^T v_i = 1$ and $v_i^T v_j = 0$ for every $i \neq j$.

The Lagragian is:

$$F(\tilde{V}, \lambda) = \sum_{i=k+1}^{d} v_i \Sigma v_i + \sum_{i=k+1}^{d} \lambda_i (1 - v_i^T v_i) + \sum_{i=k+1}^{d} \sum_{i \neq j = k+1}^{d} \lambda_{i,j} v_i^T v_j$$

. Setting the deriviative to zero yields:

$$\frac{\partial F(\tilde{V}, \lambda)}{\partial v_i} = 2\Sigma v_i - 2\lambda_i v_i + \sum_{i \neq j = k+1}^{d} \lambda_{i,j} v_j \overset{!}{=} 0$$

When multiplying with v_j $(j \neq i)$, one sees, that $\lambda_{i,j}$ has to be zero(beacuse of the conditions (1) and (2)):
$v_j \frac{\partial F(\tilde{V},\lambda)}{\partial v_i} = \lambda_{i,j} \stackrel{!}{=} 0.$

What remains is the following:

$$\frac{\partial F(\tilde{V},\lambda)}{\partial v_i} ! = 0 \Leftrightarrow \Sigma v_i = \lambda_i v_i$$

Therefore, we know that $\tilde{V}$ has to only consist out of eigenvectors of the covariance matrix. But which ones?
We already know that the error is $L \cdot \sum_{i=k+1}^{d} \lambda_i$. From this, it immediately follows, that for $v_{k+1}, \ldots, v_d$ the eigenvectors with the smalles eigenvalues have to be chosen.

In a nutshell:

The reconstruction error of the PCA is minimal, if we choose the k eigenvectors with the biggest eigenvalues to construct $\tilde{V}$, then the following holda for the reconstruction error:

$$\sum_{l=1}^{L} \|x - \tilde{V}\tilde{V}x_l\|_2^2 = L \cdot \sum_{i=k+1}^{d} \lambda_i$$

, where $\lambda_{k+1}, \ldots, \lambda d$ are the smalles eigenvalues of the covariance matrix of our training data.

c)
When utilizing SGD, still the equation from above holds (B is the batch size):

$$\sum_{l=1}^{B} \|x - \tilde{V}\tilde{V}x_l\|_2^2 = (B-1) \cdot \sum_{i=k+1}^{d} v_i^T \hat{\Sigma} v_i$$

However, this time we only have an estimate of the the covariance matrix $\hat{\Sigma}$.
Here $\hat{\Sigma} = \frac{1}{B-1} \sum_{i=1}^{B} (x - \overline{x})(x - \overline{x})^T$. This estimator is at least unbiased, but for small B the training will sigificantly suffer. For bigger B the there shouldn't be a noticable performance loss.

[ ] of 20 Points