Closed book exam, no books, papers, notes, phones etc. allowed. The exam has 8 coding questions (labs), 8 open questions (lectures) and 14 multiple choice questions (papers). **Answer on the separate answer sheets.** Explain all answers, i.e. explicitly show intermediate steps to clarify if needed for coding / calculations / motivations / etc. Good luck!

1 Lab assignments (38pts)

1. **Question** (4pts) Implement the forward pass of a linear layer, followed by a ReLU activation function:

$$\mathbf{y} = \mathbf{x}\mathbf{W} + \mathbf{b}$$
$$\mathbf{y} = \text{Sigmoid}(\mathbf{y})$$

Here $\mathbf{x}$ has dimensions [batch_size, input_channels]. Use the code template provided below. You may use elementary operations from the PyTorch library only, i.e. no predefined layers from torch.nn. Sub-questions:

A) Placeholder for layer weight and bias (1pts)

B) Forward pass (3pts)

```
import torch
class Linear(object):
    """
    Fully connected layer.
    Args:
        in_feat: number of input features
        out_feat: number of output features
    """

    def __init__(self, in_feat, out_feat):
        super(Linear, self).__init__()
        ####################################
        # A) Define placeholder tensors for
        # layer weight and bias. The placehol-
        # der tensors should have the correct
        # dimension according to the in_feat
        # and out_feat variables.
        ####################################
        # A) Your code goes here.
        ####################################
        # END OF YOUR CODE
        ####################################

        self.init_params() # Init. parameters

    def init_params(self):
        """
        Initialize layer parameters. Sample
        weight from Gaussian distribution and
        bias uniform distribution.
        """
        self.weight = torch.randn_like(
            self.weight)
        self.bias = torch.rand_like(self.bias)

    def forward(self, x):
        """
        Forward pass of Linear layer: multiply
        input tensor by weights and add bias.
        Args:
            x: input tensor
        Returns:
            y: output tensor
        """
        ####################################
        # B) Forward pass.
        ####################################
        # B) Your code goes here.
        ####################################
        # END OF YOUR CODE
        ####################################

        return y
```

2. **Question** (4pts) Implement the backward passes for the ReLU and Sigmoid non-linearities. The non-linearities are defined as follows:

$$\text{ReLU}(\mathbf{x}) = \max(0, \mathbf{x})$$
$$\text{Sigmoid}(\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{x})}$$

You may use elementary operations from PyTorch library only, i.e. no predefined layers from torch.nn. Sub-questions:

A) ReLU backward (2pts)

B) Sigmoid backward (2pts)

```
import torch
class ReLU(object):
    """
    ReLU non-linear activation function.
    """

    def __init__(self):
        super(ReLU, self).__init__()

        # Cache forward pass variables for use
        # during backward pass.
        self.cache = None

    def forward(self, x):
        """
        Forward pass of ReLU non-linear
        activation function: y=max(0,x). Store
        input tensor as cache variable.
        Args:
            x: input tensor
        Returns:
            y: output tensor
        """

        y = torch.clamp(x, min=0)  # forward pass
        self.cache = y # update cache

        return y

    def backward(self, dupstream):
        """
        Backward pass of ReLU non-linear
        activation function: return downstream
        gradient dx.
        Args:
            dupstream: Gradient of loss with
            respect to output of this layer.
        Returns:
            dx: Gradient of loss with respect to
            input of this layer.
```

```
        """

        # Making sure that we don't modify the
        # incoming upstream gradient
        dupstream = dupstream.clone()

        ###################################
        # A) ReLU Backward pass
        ###################################
        # A) Your code goes here.
        ###################################
        # END OF YOUR CODE
        ###################################

        return dx

class Sigmoid(object):
    """
    Sigmoid non-linear activation function.
    """

    def __init__(self):
        super(Sigmoid, self).__init__()

        # Cache forward pass variables for use
        # during backward pass.
        self.cache = None

    def forward(self, x):
        """
        Forward pass of Sigmoid non-linear
        activation function: y=1/(1+exp(-x)).
        Store input tensor as cache variable.
        Args:
            x: input tensor
        Returns:
            y: output tensor
        """

        y = 1.0 / (1.0 + torch.exp(-x))
        self.cache = y # update cache

        return y

    def backward(self, dupstream):
        """
        Backward pass of Sigmoid non-linear
        activation function: return downstream
        gradient dx.

        Args:
            dupstream: Gradient of loss with
            respect to output of this layer.

        Returns:
            dx: Gradient of loss with respect to
            input of this layer.
        """

        ###################################
        # B) Sigmoid Backward pass
        ###################################
        # B) Your code goes here
        ###################################
        # END OF YOUR CODE
        ###################################

        return dx
```

3. **Question** (6pts) Implement the forward pass of a convolutional layer. You may use elementary operations from PyTorch library only, i.e. no pre-defined layers from torch.nn. Sub-questions:

   A) Placeholder for weight and bias (1pt)

   B) The corresponding input window (2pts)

   C) Forward pass for batch, in for loop (3pts)

```
import torch
class Conv2d(object):
    """
    2D convolutional layer.
    """

    def __init__(self, in_channels, out_channels,
       kernel_size, stride=1, padding=0):

        """
        Initialize the layer with given params
        Args:
            in_channels: num. of input channels
            out_channels: num. of output channels
            kernel_size: kernel size, single int
            stride: step size of conv. operation
            padding: num pixels to zero-pad input
        """

        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding

        ###################################
        # A)  Create placeholder tensors for
        # weight and bias with correct dims.
        ###################################
        # A) Your code goes here.
        ###################################
        # END OF YOUR CODE
        ###################################

        # Initialize parameters
        self.init_params()

    def init_params(self):
        """
        Initialize layer parameters. Sample
        weight from Gaussian distribution
        and initialize bias as zeros.
        """

        self.weight = torch.randn_like(
            self.weight)
        self.bias = torch.rand_like(self.bias)

    def forward(self, x):
        """
        Forward pass of convolutional layer

        Args:
            x: input tensor (N, C, H, W)

        Returns:
            y: output tensor (N, F, H', W'),
            where:
                H' = 1 + (H + 2 * padding -
                    kernel_size) / stride
                W' = 1 + (W + 2 * padding -
                    kernel_size) / stride
        """

        # Pad the input
        x_padded = torch.nn.functional.pad(
```

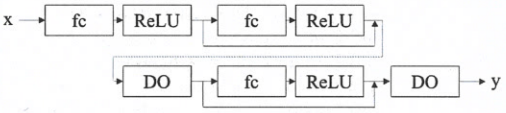
```
            x, [self.padding] * 4)

        # Unpack the needed dimensions
        N, _, H, W = x.shape

        # Calculate output height and width
        Hp = 1 + (H + 2 * self.padding -
             self.kernel_size) // self.stride
        Wp = 1 + (W + 2 * self.padding -
             self.kernel_size) // self.stride

        # Create an empty output to fill in
        y = torch.empty(
             (N, self.out_channels, Hp, Wp))


        ####################################
        # B-C) Compute the output y by looping
        # over its height and width dimensions
        # and defining an input window over x
        # called window, using which you
        # calculate the output for each input
        # sample by convolving it with the
        # weight and adding a bias.
        ####################################
        for i in range(Hp):
             for j in range(Wp):
                 # B) Make window.
                 # Your code goes here.
                 for k in range(N):
                     # C) Forward pass.
                     # Your code goes here.
        ####################################
        # END OF YOUR CODE
        ####################################

        # Cache input to use in backward pass
        self.cache = x_padded

        return y
```

4. **Question** (4pts) Implement the gradient update of SGD with momentum. The formula for a gradient update with momentum is:

$$v_i = \rho v_{i-1} + (1-\rho)\nabla_\theta$$
$$\theta' = \theta - \epsilon v_i$$

```
def momentum(X, rho, learning_rate, prev_value,
     Grad=Grad_f):
    """
    Gradient descent with momentum optimization
     step.
    Args:
       X: Current value of objective function.
       rho: Optimization hyperparameter -
            see formula above.
       learning_rate: Optimization step size.
       prev_value: Momentum parameter from
            previous iteration.
       Grad: Function that returns gradient
            of objective function.
    """
    # Gradient of current value
    gradient = Grad(*X)
    # Momentum parameter
    v = 0
    # Momentum parameter from previous iteration
    v_prev = prev_value

    ####################################
    # A) Create gradient descent with
    # momentum: update v and X.
    ####################################
    # A) Your code goes here
    ####################################
    #  END OF YOUR CODE
    ####################################

    return X, v
```

5. **Question** (4pts) Implement the Dropout layers and the forward pass of the network architecture depicted in the figure (fc := fully connected layer; DO:= dropout; merging of arrows := summation):



Sub-questions:

A) Initialize Dropout layers (1pts)

B) Implement forward pass (3pts)

```
import torch.nn as nn
class FCNet_do(nn.Module):
    """
    Simple fully connected neural network with
      residual connections and dropout
    layers in PyTorch. Layers are defined in
      __init__ and forward pass
    implemented in forward.
    """

    def __init__(self):
        super(FCNet_do, self).__init__()

        self.fc1 = nn.Linear(16, 32)
        self.fc2 = nn.Linear(32, 32)
        self.fc3 = nn.Linear(32, 64)

        ####################################
        # A) Define dropout layers.
        ####################################
        # A) Your code goes here.
        ####################################
        # END OF YOUR CODE
        ####################################

    def forward(self, x):
        ####################################
        # B) Implement forward pass as in figure
        ####################################
        # B)Your code goes here
        ####################################
        # END OF YOUR CODE
        ####################################
        return y
```

6. **Question** (4pts) Implement the vanilla RNN. [Hint]: think about all the architectural hyperparameters and especially sizes of the input, hidden states, weights and outputs. Sub-questions:

A) Parameter initialization (2pts)

B) Forward pass implementation (2pts)

```
class VanillaRNN(nn.Module):
```

```
    """
    Vanilla recurrent neural network which
    has the following update rule:
        ht = tanh(W_xh * xt + b_xh + W_hh *
            h(t-1) + b_hh)
    """
    def __init__(self, input_size, hidden_size):
        super(VanillaRNN, self).__init__()

        self.hidden_size = hidden_size

        ####################################
        # A) Create weight and bias tensors
        # with correct sizes. NOTE: Don't
        # forget to encapsulate weights and
        # biases in nn.Parameter.
        ####################################
        # A) Finish the following lines
        self.weight_xh = # input-to-hidden w
        self.weight_hh = # hidden-to-hidden w
        self.bias_xh = # input-to-hidden b
        self.bias_hh = # hidden-to-hidden b
        ####################################
        # END OF YOUR CODE
        ####################################

    def forward(self, x):
        """
        Args:
            x: input with shape (N, T, D):
                N: number of samples
                T: number of time steps
                D: input size, equal to
                    self.input_size.
        Returns:
            y: output with shape (N, T, H):
                H: hidden size
        """

        # Transpose input for efficient
        # vectorized calculation. After
        # transposing # the input will have
        # shape (T, N, D).
        x = x.transpose(0, 1)

        # Unpack dimensions
        T, N = x.shape[0], x.shape[1]

        # Initialize hidden states to zero.
        h0 = torch.zeros(N, self.hidden_size)

        # Define a list to store outputs.
        y = []

        ####################################
        # B) Implement the RNN forward pass
        ####################################
        # B) Your code goes here.
        ####################################
        # OF YOUR CODE
        ####################################

        # Stack outputs to shape (T, N, H)
        y = torch.stack(y)

        # Switch time and batch dimension
        # (T, N, H) -> (N, T, H)
        y = y.transpose(0, 1)

        return y
```

7. **Question** (6pts) Implement the basic self-attention layer in PyTorch. Recall that the input serve three different purposes in the self-attention operation, namely as the query, key and value:

$$\mathbf{q}_i = \mathbf{W}_q \mathbf{x}_i, \quad \mathbf{k}_i = \mathbf{W}_k \mathbf{x}_i, \quad \mathbf{v}_i = \mathbf{W}_v \mathbf{x}_i$$

And the basic self-attention operation:

$$w'_{ij} = \frac{\mathbf{q}_i^T \mathbf{k}_j}{\sqrt{k}}, \quad w_{ij} = \text{softmax}\left(w'_{ij}\right),$$

$$\mathbf{y}_i = \sum_j w_{ij} \mathbf{v}_j$$

Sub-questions:

A) Transform input tensor to queries, keys and values (2pts)

B) Compute weights, scale weights, apply softmax (2pts)

C) Compute output tensor (2pts)

```
import torch.nn as nn
class SelfAttention(nn.Module):
    """
    Self-attention operation with learnable key,
    query and value embeddings.
    Args:
        d: embedding dimension
    """
    def __init__(self, k):
        super(SelfAttention, self).__init__()

        # Linear map to queries, keys, values
        self.k = nn.Linear(d, d, bias=False)
        self.q = nn.Linear(d, d, bias=False)
        self.v = nn.Linear(d, d, bias=False)

    def forward(self, x):

        # Get tensor dimensions: batch size,
        # sequence length and embedding
        # dimension.
        b, t, d = x.size()

        ####################################
        # A) Perform self-attention operation
        # with learnable query, key and value
        # mappings.
        # B) Calculate w_prime, apply scaling
        # and softmax.
        # C) Compute the output tensor y.
        ####################################
        # Your code goes here.
        ####################################
        # END OF YOUR CODE
        ####################################

        return y
```

8. **Question** (6pts) Implement the auto-encoder. The whole auto-encoder is implemented in the code block below, which consists of 3 classes: Encoder, Decoder and Autoencoder. Sub-questions:

   A) Create necessary layers for Encoder (2pts)

   B) Finish forward pass of Encoder (1pts)

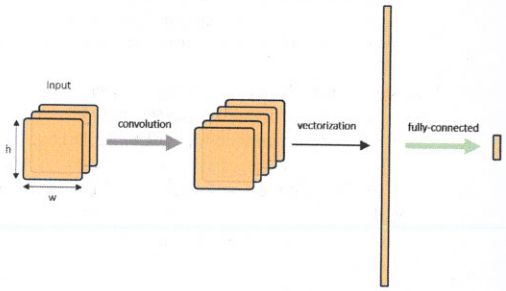
   C) Implement forward pass of Decoder (2pts)

   D) Implement forward pass of Autoencoder (1pts)

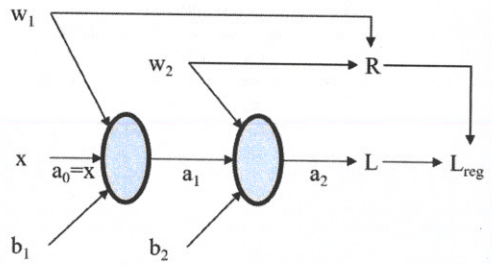
```python
import torch.nn as nn
class Encoder(nn.Module):
    """
    Encoder, projects input to latent space.
    Args:
        l_dim: latent space dimensionality (int)
        s_img: size of square input image (int)
        h_dim: dim. of hidden layers (list)
    """
    def __init__(self, l_dim, s_img, h_dim):
        super(Encoder, self).__init__()

        ######################################
        # A) Create the necessary layers
        ######################################
        # A) Your code goes here.
        ######################################
        # END OF YOUR CODE
        ######################################


    def forward(self, x):

        x = torch.flatten(x, start_dim=1)
        x = self.relu(self.linear1(x))
        x = self.relu(self.linear2(x))

        ######################################
        # B) Apply final layer of the encoder
        # [Hint]: do you need a non-linearity?
        ######################################
        # B) Your code goes here.
        ######################################
        # END OF YOUR CODE
        ######################################

        return x

class Decoder(nn.Module):
    """
    Decoder, projects latent space to image.
    Args:
        l_dim: latent space dimensionality (int)
        s_img: size of square input image (int)
        h_dim: dim. of hidden layers (list)
    """
    def __init__(self, l_dim, s_img, h_dim):
        super(Decoder, self).__init__()

        self.linear1 = nn.Linear(l_dim,
                                 h_dim[1])
        self.linear2 = nn.Linear(h_dim[1],
                                 h_dim[0])
        self.linear3 = nn.Linear(h_dim[0],
                                 s_img*s_img)
        self.relu    = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, z):

        ######################################
        # C) Implement forward pass of decoder
        # [Hint]: Have a close look at the
        # forward pass of the encoder.
        ######################################
        # C) Your code goes here.
        ######################################
        # END OF YOUR CODE
        ######################################

        return z

class Autoencoder(nn.Module):
    """
    Autoencoder model.
    Args:
        l_dim: latent space dimensionality (int)
        s_img: size of square input image (int)
        h_dim: dim. of hidden layers (list)
    """
    def __init__(self, l_dim, s_img,
                 h_dim = [256, 128]):
        super(Autoencoder, self).__init__()

        self.encoder = Encoder(l_dim, s_img,
    h_dim)
        self.decoder = Decoder(l_dim, s_img,
    h_dim)

    def forward(self, x):

        ######################################
        # D) Autoencoder forward pass.
        ######################################
        # D) Your code goes here.
        ######################################
        # END OF YOUR CODE
        ######################################

        return y
```

2 Lectures (38pt)

1. **Question** (4pts) Consider a 2-layered network. The first layer is convolutional with a kernel size of 3×3, with padding = 1, stride = 1. The second layer is a fully connected layer. The number of input channels is 3, the number of hidden channels is 5. The amount of output features is 2. The network uses bias terms. The size of an input image is: $h \times w = 16 \times 16$. The image below clarifies the network further. **What is the total amount of learnable parameters?** Motivate your answer with a detailed step-by-step approach. (Don't forget the bias terms).



2. **Question** (6pts) Consider the following scalar feedforward neural network with input x and target y.



$\mathrm{L} = (a_2 - y)^2$ and $\mathrm{R} = \sum_i w_i^2$ are the loss and regularization term respectively and the regularized loss is computed as $\mathrm{L_{reg}} = \mathrm{L} + \lambda \mathrm{R}$, in which λ is some predefined weighting hyperparameter. Furthermore the activations a_i are computed as

$$a_i = \sigma(z_i) = \sigma(w_i a_{i-1} + b_i),$$

in which σ is some arbitrary continuously differentiable non-linear activation function (e.g. sigmoid, tanh). We are interested in computing $\frac{\partial \mathrm{L}_{reg}}{\partial w_1}$. We have already computed $\frac{\partial \mathrm{L}_{reg}}{\partial w_2}$ and $\frac{\partial \mathrm{L}_{reg}}{\partial b_2}$ for which we used $\overline{z_2} = \frac{\partial \mathrm{L}_{reg}}{\partial z_2}$. Efficient backpropagation uses this intermediate result for the computation of gradients of earlier layers. Compute $\frac{\partial \mathbf{L}_{reg}}{\partial w_1}$ based on this result. Motivate your answer (do not only write down the formula). If needed, use $\sigma'()$ for the derivative of the non-linear activation function.

3. **Question** (6pts) We would like to perform a convolution on an $5 \times 5 \times 2$ image (ie: 2 channels, so 2 values per pixel) with an "average value kernel with horizontal and vertical spatial size 3" (no padding) and a stride of 1. The value of the input image X is given below:

$$X[:,:,0] = \begin{bmatrix} 1 & 0 & 2 & 6 & 4 \\ 0 & 2 & 1 & 2 & 5 \\ 2 & 1 & 0 & 4 & 2 \\ 6 & 2 & 4 & -4 & -3 \\ -4 & -2 & 0 & 3 & -2 \end{bmatrix}$$

$$X[:,:,1] = \begin{bmatrix} 0 & -2 & 1 & 2 & 2 \\ -3 & 0 & -2 & 3 & 2 \\ 1 & -2 & -2 & 2 & 2 \\ 0 & 4 & 4 & -1 & -1 \\ 2 & 2 & 0 & 3 & -2 \end{bmatrix}$$

(2pt) 1. Give the kernel matrix.
(4pt) 2. Compute output of the convolution.

4. **Question** (4pts) About regularization.

(1pt) 1. What is underfitting?
(1pt) 2. What is overfitting?
(1pt) 3. How to detect underfitting?
(1pt) 4. How to detect overfitting?

5. **Question** (6pts) The exponentially weighted moving average (EWMA) is used in many optimization methods. The equation is given by: $S_t = (\rho S_{t-1}) + (1 - \rho) y_t$.

Consider a case where the gradient has a constant value of 1.0, over all considered training samples. Let $\rho = 0.95$.

(2pt) 1. Compute S_2.

The bias correction equation is: $\hat{S}_t = \frac{S_t}{1-\rho^t}$.

(2pt) 2. Compute the bias corrected $\hat{S}_2$.

(1pt) 3. What is the goal of bias correction?

(1pt) 4. What role does bias correction play after a large number of iterations?

6. **Question** (4pts) Can you explain why vanishing/exploding gradients happen in RNNs? What changes can be made to RNNs to solve these problems?

7. **Question** (4pts) Regarding self-attention.

(1pt) 1. What is the computational advantage of self-attention over recurrent networks?
(1pt) 2. What is the purpose of having multiple attention heads?
(1pt) 3. What is the difference between narrow and wide multi-head self-attention?
(1pt) 4. How does the number of parameters in a self-attention layer change if the input sequence length doubles?

8. **Question** (4pts) Explain the GAN objective function given below:

$$\arg\min_{\theta_G}\max_{\theta_D}\mathbb{E}_{x\sim p_{\text{data}}}\log D_{\theta_D}(x) + \mathbb{E}_{z\sim p(z)}\log\left(1-D_{\theta_D}\left(G_{\theta_G}(z)\right)\right)$$

3 Papers (14pts)

Each paper has a multiple-choice question. Select the single best fitting answer per question. Each question counts for 1 point.

1. Paper: *A Step Toward Quantifying Independently Reproducible Machine Learning Research.* What statement is mentioned in the paper:
   - A) Without releasing code it is very difficult to reproduce ML research.
   - B) Code should be reviewed in the scientific peer-review process.
   - C) Its OK if approximately 25% of the claims in a paper cannot be reproduced.
   - D) Reproducability should become part of university courses.
2. Paper: *Troubling Trends in Machine Learning Scholarship.* The paper mentions "Explanation vs. Speculation". What exactly is meant by that?
   - A) Papers are intentionally made overly complex.
   - B) There is too much math
   - C) There is not sufficient explanation for some claims
   - D) The method is not explained.
3. Paper: *Do ImageNet Classifiers Generalize to ImageNet?* So, do they generalize?
   - A) Yes, although the accuracy is reduced, the ranking between models stays the same
   - B) Yes, because it showed to generalize to other, slightly changed, domains.
   - C) No, because the technical term "generalization" only applies to the original data
   - D) No, because there was insufficient data
4. Paper: *Scaling down deep learning.* According to the paper, why scale down?
   - A) Complex models are too difficult to interpret
   - B) Results on small-scale datasets often translate to large-scale datasets
   - C) Scaling down is the only option for smaller labs and universities
   - D) Scaling down is essential for scientific reproductions
5. Paper: *Highway and Residual Networks learn Unrolled Iterative Estimation.* What best describes a residual connection?
   - A) Add the output of the previous layer to the input of the current layer
   - B) Add the output of the previous layer to the output of the current layer
   - C) Add the input of the previous layer to the output of the current layer
   - D) Add the input of the previous layer to the input of the current layer
6. Paper: *ResNet strikes back: An improved training procedure in timm* .
   - A) An independent reproduction of a standard ResNet with an improved training procedure improves over the original paper
   - B) An independent reproduction of a modified ResNet with an improved training procedure improves over the original paper
   - C) An independent reproduction of a standard ResNet with an improved training procedure cannot improve over the original paper
   - D) An independent reproduction of a modified ResNet with an improved training procedure cannot improve over the original paper
7. Paper: *Deep Image Prior.* What statement fits best with the paper:
   - A) Neural network architectures are a form of prior knowledge
   - B) ConvNets are a form of prior knowledge for images
   - C) Optimization is less important as the network architecture
   - D) ConvNets can learn to turn noise to images
8. Paper: *Approximating CNNs with Bag-of-local-Features models works surprisingly well on ImageNet.* Why is it called a BagNet?
   - A) Its based on unordered bagging and boosting bootstrapping
   - B) The receptive field is artificially unordered
   - C) The class evidence used for explainability is bagged across unordered classes
   - D) It treats images as a unordered bag of patches

9. Paper: *Group Normalization.* The statistics are computed over groups of pixels. The grouping is done by:
   A) Grouping spatially
   B) Grouping feature maps
   C) Grouping images
   D) Grouping weights
10. Paper: *Torch.manual_seed(3407) is all you need: On the influence of random seeds in deep learning architectures for computer vision.*
   A) Black swans can be found, although the variance is small
   B) Black swans cannot be found because the variance is small
   C) Black swans can be found as the variance is large
   D) Black swans cannot be found, although the variance is large
11. Paper: *Attention Is All You Need.* Select the best answer.
   A) The query, key and value are shared over the inputs, similar to a 1x1 CNN
   B) The connections between the network layers are gated, similar to a GRU
   C) In a self-attention layer, information flows sequentially between tokens, similar to an LSTM.
   D) Without positional embeddings, it would overfit on the word position, similar to a MLP.
12. Paper: *Perceiver IO: A General Architecture for Structured Inputs & Outputs.*
   A) Uses standard self-attention, and it is outperformed by specialized solutions
   B) Uses a special form of self-attention, and it is outperformed by specialized solutions
   C) Uses standard self-attention and it outperforms specialized solutions
   D) Uses a special form of self-attention and it outperforms specialized solutions
13. Paper: *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks.* What does the "cycle" refer to?
   A) Cyclic activation functions
   B) Training cycles
   C) Cycling between the annotated image pairs
   D) Cycling between domains
14. Paper: *GANORCON: Are Generative Models Useful for Few-shot Segmentation?* So, are GANs useful for few-show segmentation?
   A) Yes, more useful than contrastive learning.
   B) Yes, but not as useful as contrastive learning
   C) No, but contrastive learning is
   D) No, and contrastive learning also is not