Freie Universität Berlin
FB Mathematik und Informatik
Prof. Dr. Frank Noé

SoSe 2020
03.08. - 07.08.2020 (1pm CET)

Deep Learning (19238501)

Exam

Name, given name: _____

Matriculation number: _____

Institution: _____

I hereby declare that I have done the work this exam independently and without help from others, and that I have not communicated my solution to other students. I am aware that clearly plagiarized solutions will lead to exams graded with 0 points and if two or more submissions contain clear duplicates beyond a reasonable probability of coincidence, all exams containing duplicate solutions will be graded with 0 points.

Signature: _____

Make sure all notes are clearly assigned to the relevant **number of the examination task**!

Do not hesitate to ask any of the lecture personnel per E-mail (jonas.koehler@fu-berlin.de or leon.klein@fu-berlin.de or moritz.hoffmann@fu-berlin.de) should you have any exam-related questions whatsoever!

Question:	I	II	III	IV	Total
Points:	14	8	14	11	47
Score:					

Grade:

Examiner:

I. Multiple choice

The multiple-choice questions can have one or more correct answers. You are expected to explain your answers in one or two sentences.

(A) Consider a neural network with a two-dimensional input and a one-dimensional output, i.e., no hidden layers, two weights (w_1 and w_2) and one bias (b). The activation of the output node is given by a sigmoid function. You want to represent the following table approximately: (2 pts)

Input 1	Input 2	Output
1	1	0
1	0	0.5
0	1	0.5
0	0	1

How do you choose the two weights and the bias?

- ■ $w_1 = -10$, $w_2 = -10$, $b = 10$
- ☐ $w_1 = 10$, $w_2 = -10$, $b = -10$
- ☐ $w_1 = -10$, $w_2 = 10$, $b = 10$
- ☐ $w_1 = 10$, $w_2 = 10$, $b = -10$

(B) Consider a trained neural network with 3 input nodes connected to a single output node with an unknown activation function. The weights of this network are $(w_1, w_2, w_3) = (7.3, 193.4, -30.2)$ and the bias is $b = 8.3$. Given the input vector $(i_1, i_2, i_3) = (2.1, 5.2, -0.2)$ you observe $out = 1.0$ as an output (rounded to one decimal place). Which activation functions could have been used? (2 pts)

- ☐ ReLU
- ■ **Tanh**
- ■ **Sigmoid**
- ☐ Leaky ReLU

(C) You can choose between two network architectures for a binary classification problem. The only difference between the two models is the output layer, as the first model has only one output neuron and the second two. Which of these networks can be trained successfully? (2 pts)

- ■ **The model with one output neuron**
- ■ **The model with two output neurons**
- ☐ Neither of them

(D) Consider a dense neural network with one dense hidden layer with 512 neurons and an output layer with 10 output neurons. The network is used to classify grayscale images with 28×28 pixels. How many trainable parameters do we have (including biases)? (2 pts)

- ☐ $512 \times 10 + 28 \times 512$
- ■ $512 \times 10 + 28 \times 28 \times 512 + 512 + 10$
- ☐ $512 \times 10 + 28 \times 28 \times 512$
- ☐ $512 \times 10 + 28 \times 512 + 512 + 10$

(E) Consider a single layer of a convolutional network. Given the input matrix $\mathbf{A}$ of shape 3×5 and the kernel $\mathbf{K}$ of shape 3×3. What will be the output of applying the kernel with stride 2 (valid padding)? (2 pts)

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

$$\mathbf{K} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

- ■ $\begin{bmatrix} -1 & 4 \end{bmatrix}$
- ☐ $\begin{bmatrix} -1 & -3 & 4 \end{bmatrix}$
- ☐ $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$
- ☐ $\begin{bmatrix} 4 & -2 & 4 & 3 & -1 \\ 4 & -1 & -3 & 4 & -1 \\ -1 & -1 & 5 & -2 & 0 \end{bmatrix}$

(F) Given the covariance matrix of some data (2 pts)

$$\mathbf{C} = \mathbf{X}^T\mathbf{X} = \begin{bmatrix} 2.5, & 5. & 0.5 \\ 5. & 10. & 1. \\ 0.5 & 1. & 5. \end{bmatrix}.$$

After performing a principal component analysis, how many uncorrelated dimensions of the data do you expect?

- ☐ 1
- ■ **2**
- ☐ 3
- ☐ 0

(G) Consider a convolution kernel of size 5×5, a stride $(2, 2)$, and an input image of size 25×25 grayscale pixels. What is the output dimension after applying the kernel using valid padding? (2 pts)

- ☐ 5×5
- ☐ 25×25
- ☐ 32×64
- ■ 11×11

II. Small questions

The multiple-choice questions can have one or more correct answers. You are expected to explain your answers.

(A) Consider a simple convolutional neural network which performs the following operation (3 pts)

$$\text{out} = \text{MaxPool}_{2\times2}(\text{ReLU}\,(\text{Conv}(x))),$$

where x is the input the network and out the output. Conv denotes a 2D convolution layer with a single input and output channel, $\text{MaxPool}_{2\times2}$ a max-pooling layer with a 2×2 filter and stride 2, and ReLU denotes the non-linearity ReLU. The input are grayscale 2D images, where the intesity of each pixel is given by a value between 0 (black) and 1 (white).
Which of the following are equivalent descriptions of the forward pass of the network?

- ☐ $\text{out} = \text{Conv}(\text{MaxPool}_{2\times2}(\text{ReLU}(x)))$
- ☐ $\text{out} = \text{MaxPool}_{2\times2}(\text{Conv}(\text{ReLU}(x)))$
- ■ $\mathbf{out = ReLU(MaxPool}_{2\times2}\,\mathbf{(Conv}(x)))$
- ☐ $\text{out} = \text{ReLU}(\text{Conv}(\text{MaxPool}_{2\times2}(x)))$

Does the answer change if all parameters of the kernel in the convolution layer are positive?

Solution:

- $\text{MaxPool}_{2\times2}$ and Conv do not commute.
- $\text{MaxPool}_{2\times2}$ and ReLU commute.
- Conv and ReLU commute if all parameters are positive, because ReLU is the identity for positive values. $\text{out} = \text{MaxPool}_{2\times2}(\text{Conv}(\text{ReLU}(x)))$ is also equivalent in that case.

(B) You are given data of car sizes and their CO2 emissions. Your intern has already sorted the data by car sizes and split the data into train and validation set. You want to use these data sets to train a neural network that predicts the CO2 emission based on car sizes. However, you observe that your model always performs poorly on the validation set regardless of the model architecture. What is most likely the reason for this behavior? We assume that the CO2 emission is approximately linear to the car size. (3 pts)

Solution: The data is sorted. Hence, the train and validation set stem from different distributions.

What would be a good alternative to the neural network?

Solution: Linear regression

(C) You want to train a neural network with stochastic gradient descend. However, to speed up the training by parallelization you decide to use the whole training set in each iteration. You quickly realize that the training loss does not decrease after a couple of iterations, while the validation loss is still very high. How can this behavior be explained? (2 pts)

Solution:

- Not stochastic, because the whole data set is used.
- Ordinary gradient descent gets stuck in local minima.

III. Particle system property prediction

You are given a data set consisting of $N = 500$ data points, where each data point is a set of 20 particles in three-dimensional space, i.e.,

$$X_{\text{train}} \in \mathbb{R}^{N\times 20\times 3}.$$

For each of the data points you have a scalar property that can be computed from the spatial configuration of the particles, i.e.,

$$P(X_{\text{train},i}) = y_{\text{train,i}} \in \mathbb{R} \quad \text{for all } i = 1,\dots,N,$$

with $X_{\text{train}} = (X_{\text{train},1},\dots,X_{\text{train},N})$ and $Y_{\text{train}} = (y_{\text{train},1},\dots,y_{\text{train},N})$.

The goal of this task is to find and compare neural network architectures F which try to predict $F(X) \approx P(X) = y$ for $X \in \mathbb{R}^{20\times 3}$ and $y \in \mathbb{R}$.

The property $P(X)$ that should be predicted is invariant to rotations of the system, i.e., for each rotation matrix $R \in \mathbb{R}^{3\times 3}$, it is

$$P\left(\begin{pmatrix} R\mathbf{x}_1 & R\mathbf{x}_2 & \cdots & R\mathbf{x}_{20}\end{pmatrix}^\top\right) = P(X) = y.$$

Also the particles can be interchanged with one another, the sequence in which they are enumerated does not matter: If $\mathbf{x}_j \in \mathbb{R}^3$ is the j-th particle of $X \in \mathbb{R}^{20\times 3}$, then for any permutation $\pi : \{1,\dots,20\} \to \{1,\dots,20\}$ it is

$$P\left(\begin{pmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_{20}\end{pmatrix}^\top\right) = P\left(\begin{pmatrix} \mathbf{x}_{\pi(1)} & \mathbf{x}_{\pi(2)} & \cdots & \mathbf{x}_{\pi(20)}\end{pmatrix}^\top\right) = y.$$

Please note that a permutation is by definition one-to-one: Reorderings which assign two indices to the same index and/or omit elements in the image space, e.g., $\pi(3) = \pi(7) = 1$, are **not** allowed.

(A) Implement F (**using PyTorch and NumPy API**) as a multilayer perceptron with four hidden layers and ELU (Exponential Linear Unit) nonlinearities by reshaping the data tensor to $X \in \mathbb{R}^{N\times 60}$. Structure your solution code as follows: (2 pts)

```
import torch
import torch.nn as nn
import numpy as np

class MLP(nn.Module):

    def __init__(self):
        super().__init__()
        #  ... your code here ...

    def forward(self, inputs):
        # inputs is a torch tensor with shape (batch_size, 60)
        # ... your code here ...
```

Solution:

```
import torch
import torch.nn as nn

class MLP(nn.Module):

    def __init__(self):
        super().__init__()
        units=[60, 128, 128, 64, 32, 1]  # 4 hidden layers, 1 output layer
        layers = []
        for fan_in, fan_out in zip(units[:-2], units[1:-1]):
            layers.append(nn.Linear(fan_in, fan_out))
            layers.append(nn.ELU(inplace=True))
        layers.append(nn.Linear(units[-2], units[-1]))
        self._seq = nn.Sequential(*layers)

    def forward(self, data):
        return self._seq(data.view(len(data), -1))
```

(B) Implement batchwise data augmentation routines (**using PyTorch and NumPy API**) as follows: (3 pts)

```
def prepare_batch(data, augment_rotation: bool, augment_permutation: bool):
    # data is a numpy array with shape (batch_size, 20, 3)
    out = data.copy()
    if augment_rotation:
        # ... your code here ...
    if augment_permutation:
        # ... your code here ...
    # return augmented data with shape (batch_size, 60)
    return out
```

Hint: You can sample a random rotation matrix by performing a QR-decomposition on a matrix $M \in \mathbb{R}^{3\times3}$ with normal distributed elements $M_{ij} \sim \mathcal{N}(0, 1)$:

```
import numpy as np
M = np.random.normal(size=(3, 3))
rotation_matrix = np.linalg.qr(M)[0]
```

For permutation augmentation the function `np.random.shuffle` might be useful.

Solution:

```
import numpy as np

def prepare_batch(data, augment_rotation: bool, augment_permutation: bool):
    # data is a numpy array with shape (batch_size, 20, 3)
    out = data.copy()
    if augment_rotation:
        z = np.random.normal(size=(len(out), 3, 3))
        for t in range(len(out)):
            rotmat = np.linalg.qr(z[t])[0]
            out[t] = out[t] @ rotmat.T
    if augment_permutation:
        for t in range(len(out)):
            np.random.shuffle(out[t])
    # return augmented data with shape (batch_size, 60)
    return out.reshape(-1, 60)
```

(C) Imagine implementing F using an architecture which is independent of particle permutations by design. This can be achieved by using two independent neural-network blocks ρ and φ, so that (1 pt)

$$F(X) = \rho\left(\sum_{i=1}^{20} \varphi(\mathbf{x}_i)\right), \quad X \in \mathbb{R}^{20\times3}. \tag{1}$$

Give a short argument why Equation (1) is invariant under permutations of atoms.

Solution: The function φ is shared among particles and sums commute.

(D) Provide an implementation (**using PyTorch and NumPy API**) of the architecture described in (C). In the implementation, the network φ should have three hidden layers and the network ρ should have two hidden layers. Both networks should use ELU (Exponential Linear Unit) nonlinearities. Use the following code structure: (5 pts)

```
import torch
import torch.nn as nn
import numpy as np

class Phi(nn.Module):

    def __init__(self):
        super().__init__()
        # ... your code here ...
    def forward(self, inputs):
        # ... your code here ...

class Rho(nn.Module):

    def __init__(self):
        super().__init__()
        # ... your code here ...
    def forward(self, inputs):
        # ... your code here ...

class F(nn.Module):

    def __init__(self):
        super().__init__()
        # ... your code here ...
    def forward(self, inputs):
        # inputs is a torch tensor of shape (batch_size, 20, 3)
        # ... your code here ...
```

Hint: The linear layer of PyTorch can also handle tensors with more than two dimensions (`len(data.shape)>2`):

```
from torch.nn import Linear
import numpy as np

lin = Linear(32, 16)
data = np.random.normal(size=(100, 20, 32))
with torch.no_grad():
    out = lin(torch.from_numpy(data.astype(np.float32)))
```

is the same as computing

```
out = np.empty((100, 20, 16))
for i in range(100):
    for j in range(20):
        out[i,j] = data[i,j] @ A.T + b
```

where "@" is a matrix multiplication, $A \in \mathbb{R}^{32\times16}$ the weight matrix, and $b \in \mathbb{R}^{16}$ the bias vector.

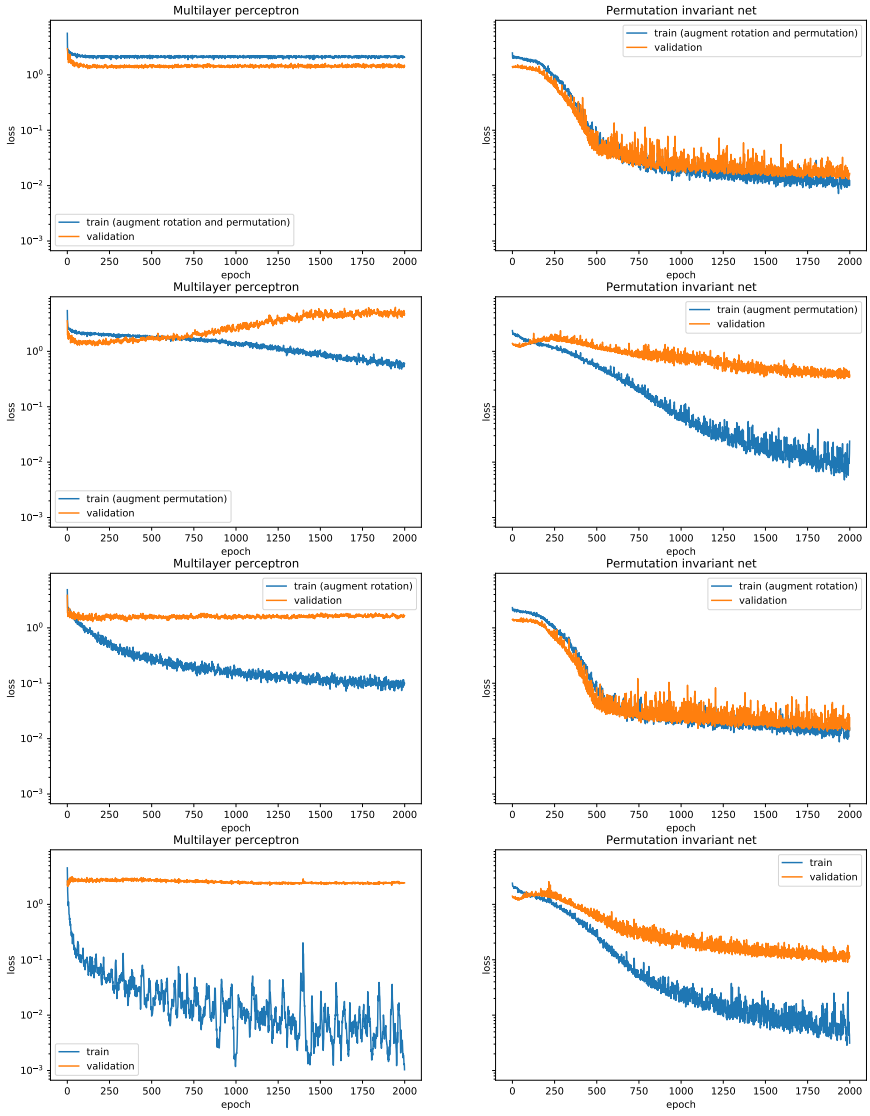
Solution:

```
import torch
import torch.nn as nn

class Phi(nn.Module):
    def __init__(self):
        super().__init__()
        self._seq = nn.Sequential(
            nn.Linear(3, 64), nn.ELU(), nn.Linear(64, 64), nn.ELU(),
            nn.Linear(64, 64), nn.ELU(), nn.Linear(64, 64), nn.ELU()
        )

    def forward(self, inputs):
        return self._seq(inputs)

class Rho(nn.Module):
    def __init__(self):
        super().__init__()
        self._seq = nn.Sequential(
            nn.Linear(64, 64), nn.ELU(), nn.Linear(64, 32), nn.ELU(),
            nn.Linear(32, 1)
        )

    def forward(self, inputs):
        return self._seq(inputs)

class F(nn.Module):
    def __init__(self):
        super().__init__()
        self.phi = Phi()
        self.rho = Rho()

    def forward(self, inputs):
        return self.rho(self.phi(inputs).sum(axis=1))
```

(E) Compare the models from (i) and (ii) by interpreting plots of training and validation curves in the cases of (2 pts)

- no data augmentation,
- augment data by applying random rotations,
- augment data by applying random permutations,
- augment data by applying random rotations and permutations.

Note that the validation data was subject to random rotations and permutations. Explain the differences.



Solution:

- Formulate a description that does not contradict itself.
- Note that permutation augmentation is a no-op for the permutation invariante net

- Note that the MLP is either over- or underfitting, depending on the state of augmentation
- Realize that MLP has poor generalization capabilities with the given data set – despite higher fitting capacity (see panel without any augmentation).
- The permutation invariant net can generalize better, especially when using rotation augmentation.
- (optional) Note that there is a fundamental difference between augmenting rotations and augmenting permutations: Rotations can be interpolated, with permutations there is combinatorial complexity.

(F) How can this observed effect be explained with the bias-variance decomposition? You don't need to prove something, as long as you are able to argue clearly. (1 pt)

Solution: Bias-variance decomposition tells us that the error decomposes into a noise term which we cannot control directly as well as bias and variance terms.

Finding a good model in the training process with limited data is finding a good balance between bias and variance. The more complex a model is (meaning the larger its accessible function space), the smaller the bias. This in particular can be observed with the MLP in the case of no data augmentation, as the training loss is very low and it is able to represent the training data very well. This low bias on the other hand means that the variance is high, i.e., the model fails to generalize.

This can also be observed for the other augmentation cases of the MLP except for the full augmentation case where it starts to underfit.

On the other hand the permutation invariant net strikes a better balance between bias and variance. This is due to the particular functional form of the network which incorporates the prior knowledge of permutational invariance (inductive bias). In particular this restricts the accessible functional space of the network to functions which are in fact permutation invariant, yielding a higher bias than the MLP.

This example showcases that it can pay off to think about the properties of what should be learnt and try to incorporate these into the design of a network, rather than regularizing or simplifying it.

IV. Backpropagation and invertible neural networks

In this task we aim to solve a regression problem with a neural network (NN) consisting of L subsequent layers. We transform inputs $\mathbf{x}$ according to

$$\mathbf{z}_1 = f_1(\mathbf{x}), \qquad \mathbf{z}_{l+1} = f_l(\mathbf{z}_l)$$

where f_l are the layers of the NN and try to minimize the expected loss

$$\mathcal{L} = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\left[\ell(\mathbf{z}_L)\right]$$

with respect to the parameters of the model using stochastic gradient descent (SGD). For simplicity we assume that inputs $\mathbf{x}$ and all activations $\mathbf{z}_l$ have same dimensionality $d \in \mathbb{N}_1 = \{1, 2, 3, \ldots\}$.

(A) Let θ_{old} be the parameters of the whole NN at a given time. Assume the batch-size $B \in \mathbb{N}_1$ and a learning rate $\alpha > 0$. How do we compute one SGD update step in order to obtain θ_{new} from θ_{old}? Please explain all necessary sub-steps explicitly in detail. (4 pts)

Solution: We sample $\mathbf{x}^{(1)}, \ldots \mathbf{x}^{(B)}$ i.i.d. from the data set. Then we transform each into $\mathbf{z}_L^{(1)}, \ldots \mathbf{z}_L^{(B)}$ (forward pass) and store them for the backpropagation step. Now we compute the sample average of the loss

$$\hat{\mathcal{L}} = \frac{1}{B}\sum_{i=0}^{B} \ell(\mathbf{z}_L^{(i)}).$$

Using the backpropagation algorithm, we now compute the estimated expected gradient w.r.t. parameters θ_{old} as

$$\nabla_{\theta_{\text{old}}}\hat{\mathcal{L}} = \nabla_{\theta_{\text{old}}}\frac{1}{B}\sum_{i=0}^{B} \ell(\mathbf{z}_L^{(i)}).$$

Here you can derive the explict weight/bias update rule for the given architecture, but was not necessary. Important detail: BP computes the gradient of the averaged loss. Not the averaged gradients. The former can be computed with $O(LBd)$ memory. The later requires storing the gradients for each point in the batch resulting in $O(LBd^2)$ memory which would not work for high-dimensional data. This finally produces the update step

$$\theta_{\text{new}} = \theta_{\text{old}} - \alpha\nabla_{\theta_{\text{old}}}\hat{\mathcal{L}}.$$

(B) We now assume that each f_l is given by a dense block. A dense transformation is defined by an affine transformation which is followed by a point-wise hyperbolic tangent as activation function (see figure 2). How much memory do we need to compute one update step if we utilize the backpropagation algorithm? Ignoring (1 pt)

constants you can express the total memory consumption in big-O notation with respect to L, B and d.

Solution: We need to save activations $z_l, \ldots, z_L$ in order to compute the weight updates at each layer. So in total $O(LBd)$.

To reduce memory constraints when training very deep nets, researchers suggested to use a new *invertible* residual block (see figure 3). Here the input to the layer $\mathbf{x} \in \mathbb{R}^d$ is first split into two parts $\mathbf{x}_1 \in \mathbb{R}^{d_1}$, $\mathbf{x}_2 \in \mathbb{R}^{d_2}$. Then both parts are manipulated to $\mathbf{y}_1 \in \mathbb{R}^{d_1}$, $\mathbf{y}_2 \in \mathbb{R}^{d_2}$. Finally, the two new components are again concatenated to produce the layer's output $\mathbf{y} = [\mathbf{y}_1, \mathbf{y}_2]$.

This block allows to compute the input activations from the output activations. Thus, you do not need to store activations for computing the weight updates during backpropagation.

(C) Show that this new block is a concatenation of residual blocks. A residual block transforms the input $\mathbf{x}$ into $g(\mathbf{x})$ for some function g and computes $\mathbf{y} = g(\mathbf{x}) + \mathbf{x}$. (2 pts)

Solution: We can write

$$\mathbf{z} = \mathbf{x} + [\mu_1(\mathbf{x}), \mathbf{0}]$$

and then

$$\mathbf{y} = \mathbf{z} + [\mathbf{0}, \mu_2(\mathbf{z})] \,.$$

(D) Show that the input activations $\mathbf{x}$ can indeed be computed from the output activations $\mathbf{y}$. (3 pts)

Solution: Knowing $\mathbf{y}_1$ you can compute μ_2. Then $\mathbf{x}_2 = \mathbf{y}_2 - \mu_2$. Now we can compute μ_1 from $\mathbf{x}_2$ which allows us to compute $\mathbf{x}_1 = \mathbf{y}_1 - \mu_1$.

(E) How much memory do we need now to compute one update step if we utilize the backpropagation algorithm? Ignoring constants you can express the total memory consumption in big-O notation with respect to L, B and d. (1 pt)

Solution: As we do not need to store activations memory requirements are now $O(Bd)$.

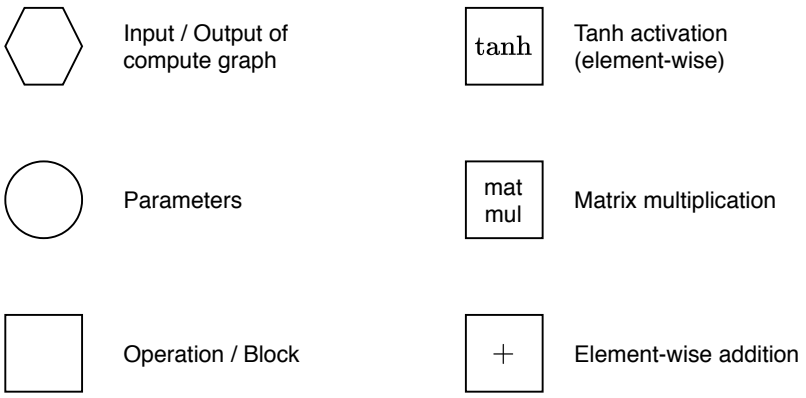


Figure 1: Graphical notation in the following figures.

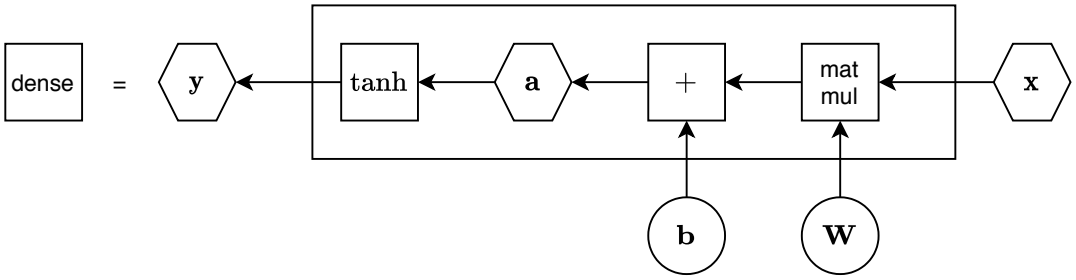


Figure 2: A dense block.

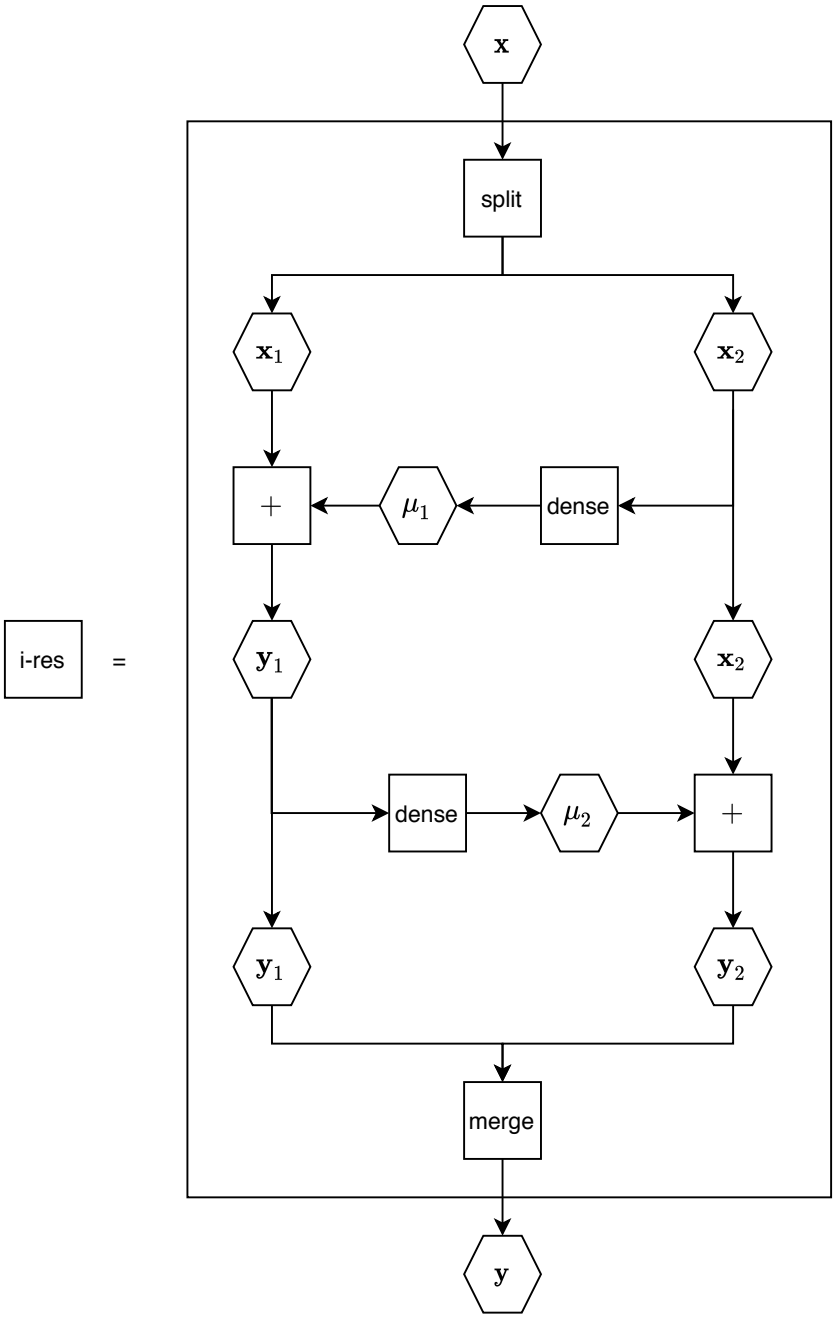


Figure 3: An invertible residual block.