AI - Lab

1. PyTorch Introduction

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Prof. Dr. Patrick Baier

SS 24

Motivation

- PyTorch is nowadays probably the most popular Deep Learning framework. (It surpassed TensorFlow in this regard).
- It is based on easy concepts that allow to build complex model architectures.
- We will introduce the concepts of PyTorch by seeing how it can replace an implementation of a Neural Network that is based on *numpy*.

What is Numpy?

- Numpy stands for „Numerical Python“.
- Numpy is a library that contains functions for fast mathematical computation with Python. For the speed-up, it relies on C-language bindings.
- Its concept is similar to Matlab, i.e. working mainly with arrays and matrices.
- As a machine learning person you should know it, since many important libraries (e.g. Sklearn, Pandas, and more) are using it under the hood.
- We will shortly introduce it, to see how PyTorch extends on it.
- Check the tutorial here.

Neural Network in Numpy

As an introduction exercise, we look at what it takes to build the following neural network architecture from scratch in Python using Numpy (the original source can be found here).

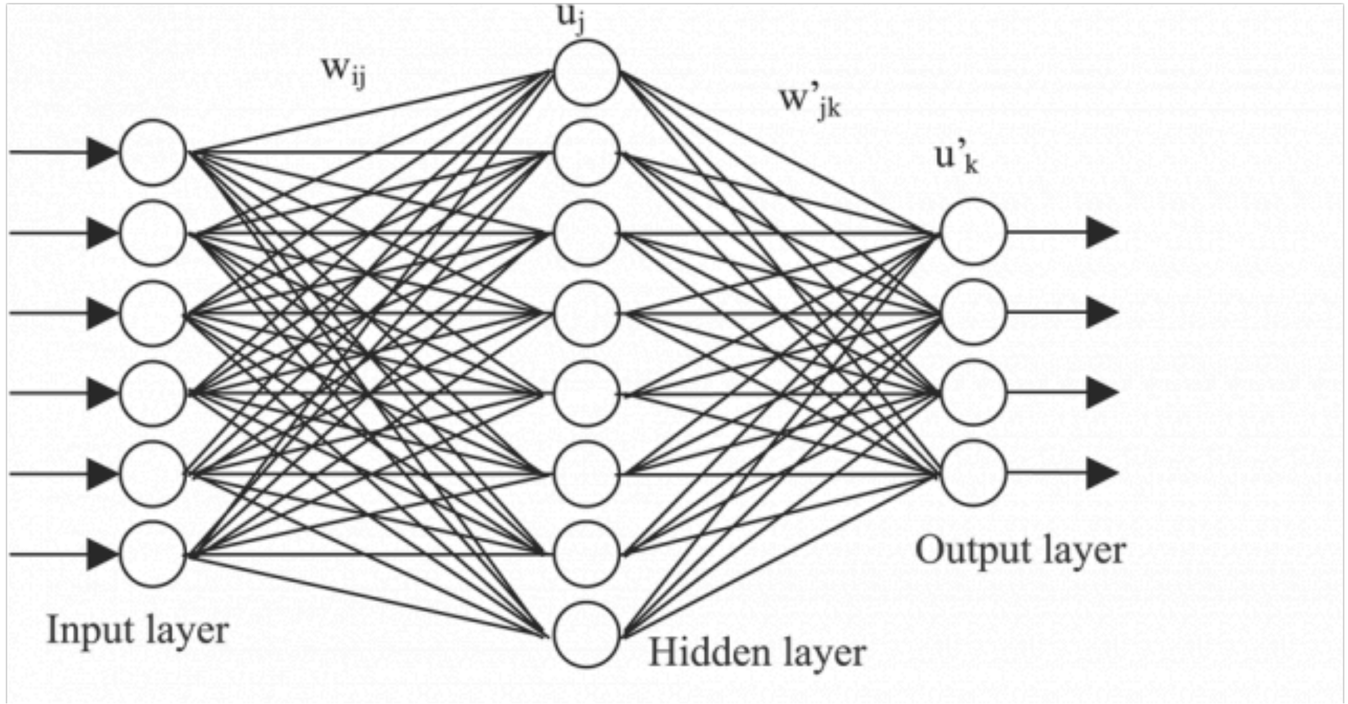
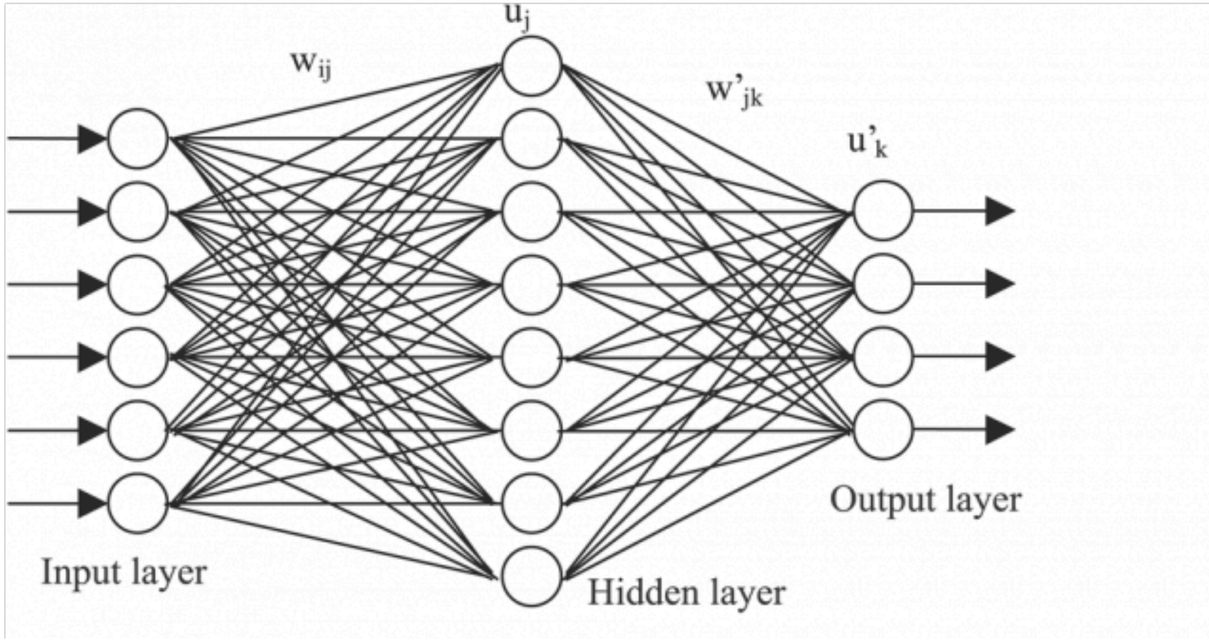


Image: https://www.extremetech.com/extreme/215170-artificial-neural-networks-are-changing-the-world-what-are-they

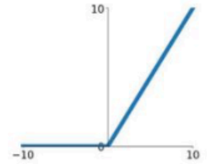
Neural Network in Numpy



- Other than in the image, we use the following dimensions:
  - Input layer: 1000 nodes
  - Hidden layer: 100 nodes
  - Output layer: 10 nodes

ReLU

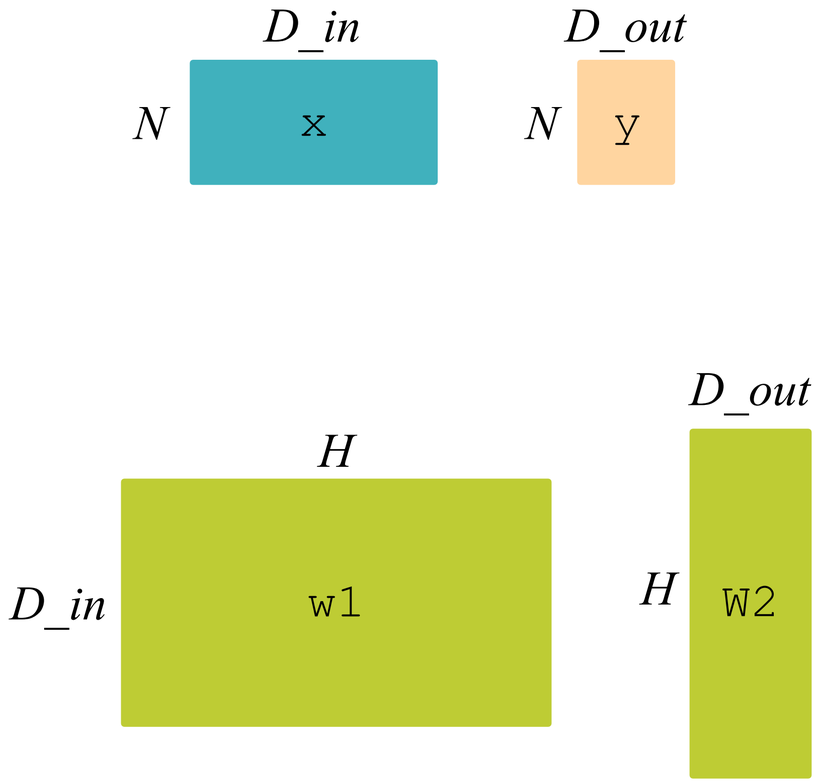
$\max(0, x)$



- We use „ReLU“ as activation function in the hidden layer.
- We use Euclidean error as loss (warning: this is not very typical).
- We use random input and random output data as training data.
- Weight parameters are learned via back propagation.

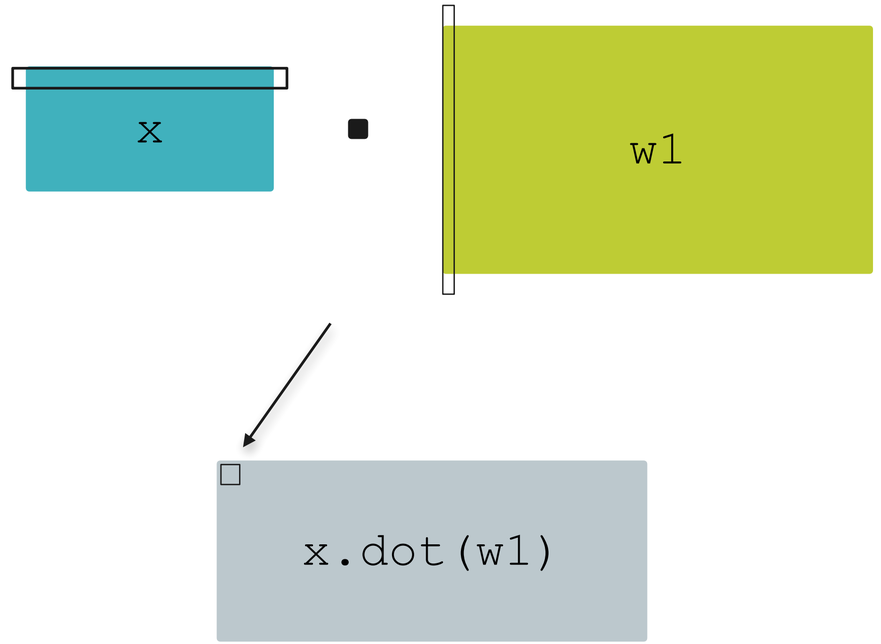
Neural Network in Numpy

```python
import numpy as np

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

# Randomly initialize weights
w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)
```



Neural Network in Numpy

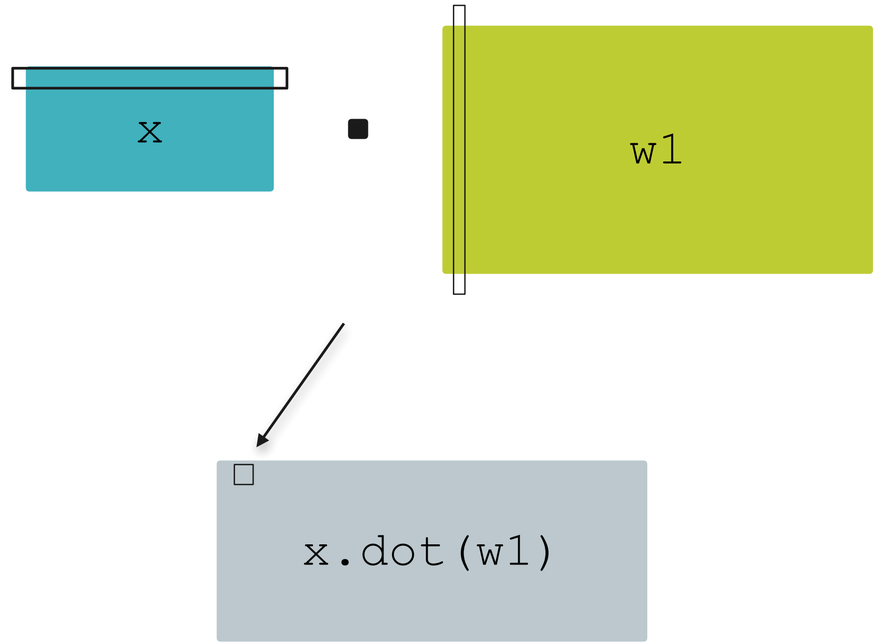
```
learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.dot(w1)
    h_relu = np.maximum(h, 0)
    y_pred = h_relu.dot(w2)

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    print(t, loss)
```



- Run training for 500 epochs.
- Forward pass is done on all data points at once (take whole `x`).
- Loss is the sum over all squared errors.

Neural Network in Numpy

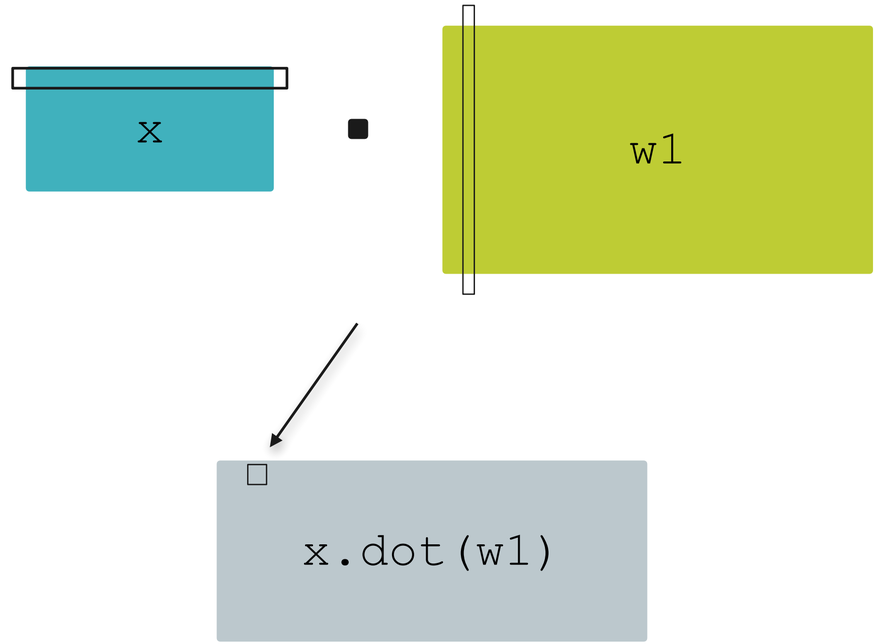
```
learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.dot(w1)
    h_relu = np.maximum(h, 0)
    y_pred = h_relu.dot(w2)

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    print(t, loss)
```



- Run training for 500 epochs.
- Forward pass is done on all data points at once (take whole `x`).
- Loss is the sum over all squared errors.

Neural Network in Numpy

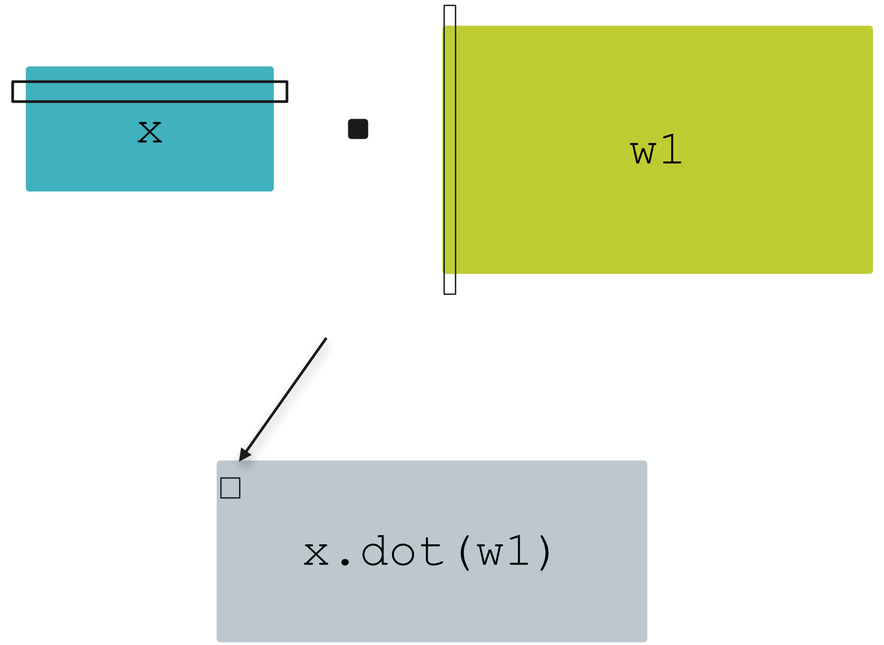
```
learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.dot(w1)
    h_relu = np.maximum(h, 0)
    y_pred = h_relu.dot(w2)

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    print(t, loss)
```



- Run training for 500 epochs.
- Forward pass is done on all data points at once (take whole `x`).
- Loss is the sum over all squared errors.

Neural Network in Numpy

```
learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.dot(w1)
    h_relu = np.maximum(h, 0)
    y_pred = h_relu.dot(w2)

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    print(t, loss)
```



- Run training for 500 epochs.
- Forward pass is done on all data points at once (take whole `x`).
- Loss is the sum over all squared errors.

Neural Network in Numpy

```
    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.T.dot(grad_y_pred)
    grad_h_relu = grad_y_pred.dot(w2.T)
    grad_h = grad_h_relu.copy()
    grad_h[h < 0] = 0
    grad_w1 = x.T.dot(grad_h)

    # Update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

- This code is also part of the loop from last slide (i.e. done in every epoch).
- Compute gradient and update weights (i.e. doing gradient descent).

Neural Network in Numpy

```
(ki-course) ➜  ~ python3 np.py
Epoch:0, loss:26311062.351246074
Epoch:50, loss:15323.53579119482
Epoch:100, loss:740.5489629434819
Epoch:150, loss:67.39831665785954
Epoch:200, loss:7.689127444872655
Epoch:250, loss:0.9645580491338863
Epoch:300, loss:0.1266611330191775
Epoch:350, loss:0.01704602978406973
Epoch:400, loss:0.0023282935849652363
Epoch:450, loss:0.00032123671398592195
```

The loss is decreasing, the network actually learns something.

Neural Network in Numpy

This was not so difficult, but also not very convenient:

- We had to implement the forward pass operations by hand.
- We had to derive the gradient computation for every layer of the network and compute it. (This will be very cumbersome for deep networks).
- And, most important, this only works on the CPU (and not on the GPU)!

Let’s see how PyTorch can help us with that.

Neural Network in PyTorch

```
import torch

dtype = torch.float
device = torch.device("cpu")
# device = torch.device("cuda:0") # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold input and outputs.
# Setting requires_grad=False indicates that we do not need to compute gradients
# with respect to these Tensors during the backward pass.
x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)
```

- We use PyTorch Tensors instead of Numpy arrays.
- Advantage: They can run automatically on the GPU!

Neural Network in PyTorch

```
# Create random Tensors for weights.
# Setting requires_grad=True indicates that we want to compute gradients with
# respect to these Tensors during the backward pass.
w1 = torch.randn(D_in, H, device=device, dtype=dtype, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, dtype=dtype, requires_grad=True)
```

- We also define the weights as a PyTorch tensors.
- We can indicate that we want to have the gradients for these tensors automatically derived.
- Advantage: We can get rid of the part in which we defined the gradients for `w1` and `w2` by ourselves.

Neural Network in PyTorch

```
learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y using operations on Tensors; these
    # are exactly the same operations we used to compute the forward pass using
    # Tensors, but we do not need to keep references to intermediate values since
    # we are not implementing the backward pass by hand.
    y_pred = x.mm(w1).clamp(min=0).mm(w2)

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())
```

- Forward pass and loss are analog to NumPy but on tensors

Neural Network in PyTorch

```
# Use autograd to compute the backward pass. This call will compute the
# gradient of loss with respect to all Tensors with requires_grad=True.
# After this call w1.grad and w2.grad will be Tensors holding the gradient
# of the loss with respect to w1 and w2 respectively.
loss.backward()
```

Here comes the magic! This call will automatically calculate the gradient with respect to `w1` and `w2`.

The basic idea behind this is to repeatedly apply the chain rule to differentiate the function. Here are some details on how this works in general and in PyTorch.

Neural Network in PyTorch

```
# Manually update weights using gradient descent. Wrap in torch.no_grad()
# because weights have requires_grad=True, but we don't need to track this
# in autograd.
# An alternative way is to operate on weight.data and weight.grad.data.
# Recall that tensor.data gives a tensor that shares the storage with
# tensor, but doesn't track history.
# You can also use torch.optim.SGD to achieve this.
with torch.no_grad():
    w1 -= learning_rate * w1.grad
    w2 -= learning_rate * w2.grad

    # Manually zero the gradients after updating weights
    w1.grad.zero_()
    w2.grad.zero_()
```

Gradients are now available and can be used for gradient descent.

Neural Network in PyTorch

- This short example already showed two of the most powerful features of PyTorch:
  1. Operations can be run on the GPU without major code changes.
  2. Support for automatic calculation of gradients on tensor operations.

- On top of this, there is a more standardized way of defining neural networks, which packs everything in a standardized format.

- On the next few slides, we see the final code for this example using the `nn` package, which contains many standard Deep Learning modules.

```python
import torch


class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        In the constructor we instantiate two nn.Linear modules and assign them as
        member variables.
        """
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        In the forward function we accept a Tensor of input data and we must return
        a Tensor of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Tensors.
        """
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred
```

```
# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# Construct our model by instantiating the class defined above
model = TwoLayerNet(D_in, H, D_out)
```

```
# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)

    # Compute and print loss
    loss = criterion(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Working Environment

- Your team can use the GPU hardware at HsKa-Lab. If you want to use that remotely there is a manual on GitHub.
- An alternative which previous courses also used: Google Colab
  - Google Log-in required (you can create a temporary one for this lab).
  - Cloud-based Jupyter notebook environment
  - Free access to GPUs (but resources are limited)
- If you have GPUs on your machine, you can also install PyTorch locally.
- Another alternative to Colab is Kaggle Kernels.

Assignment 1

Preparation:

1. Checkout the notebook `0_Simple_NN.py` from GitHub, which describes how to setup a simple feedforward network on some fake data. Try to understand everything and execute the code in a jupyter notebook.
2. Read through this blog post: https://nextjournal.com/gkoehler/pytorch-mnist
   It shows how to work with data loaders, how to load the MNIST dataset and how training is done in batches (using data loaders).

Assignment 1

Task:

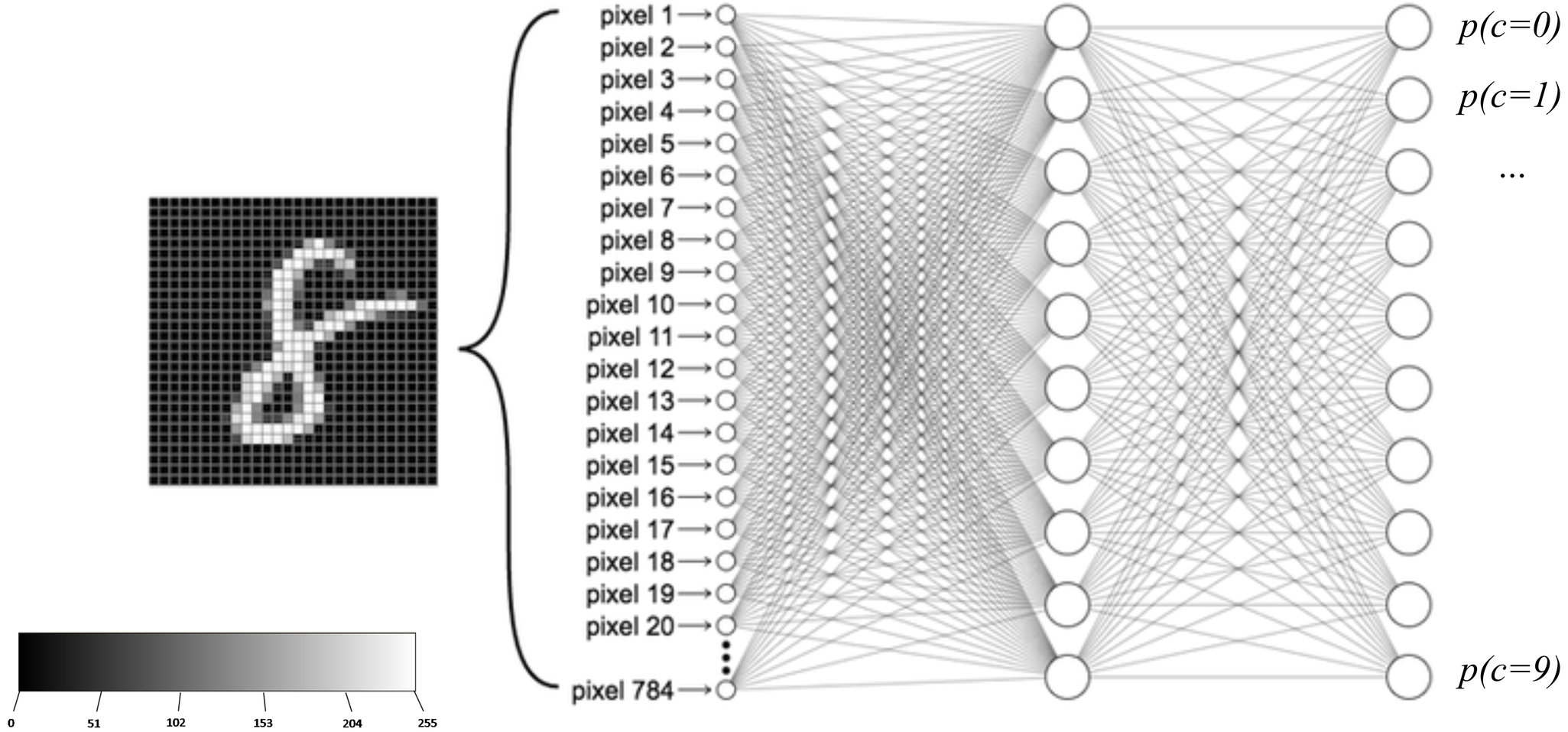
1. Load the MNIST dataset into train and test data loaders. Use the same parameters and apply the same transformations like described in the blog post.
2. Create a feedforward neural network consisting of an input layer, one hidden layer of size 100 and an output layer (same structure as in `0_Simple_NN.py`).
   For training on the MNIST dataset you need to change the following:
   - Adjust the size of the input layer to be able to take in the MNIST data (hint: you must adjust the tensor format from the MNIST data into a flat structure).
   - Use `log_softmax` as activation function for the output layer (as in the blog).
   Note: Do not use a CNN like they do in the blog post! Use Relu as activation function for the hidden layer.

Assignment 1

Task:

3. Train your network on the training data for 50 epochs using the `negative log likelihood loss` (like in the blog). Create a plot of the training loss (like in the blog but without the test loss).
4. Test the network on the MNIST test data and give out accuracy and loss.
5. Find out how the model can be trained on the GPU instead of the CPU. Compare the training time between CPU and GPU. (Note: Do not except too much improvement on this small data set).

Assignment 1



Source: https://ml4a.github.io/ml4a/looking_inside_neural_nets/

Submission Guidelines

- Send your solutions as notebook file (.ipynb) to me via e-mail, mentioning the team name and team members in the e-mail.
- Use the following e-mail subject: `[KI-LabSS24] Assignment 1 - <teamname>`
- Deadline is: **04.04.2024 - 11:59 p.m. (23:59)**

Final words

- You can start now working on assignment 1.
- It is up to you and your partners how you organize your time working on this.
- I will be around until the end of the slot at 1 pm for questions.
- Of course, questions can also be asked all time on Mattermost (but do not post any solution there!).

- Any more questions now?
- Happy Coding!