Laboratory 5

Variant #3
Mariya Zacharneva and Ruslan Melnyk

Codebase is available at lab5.ipynb.

The Task

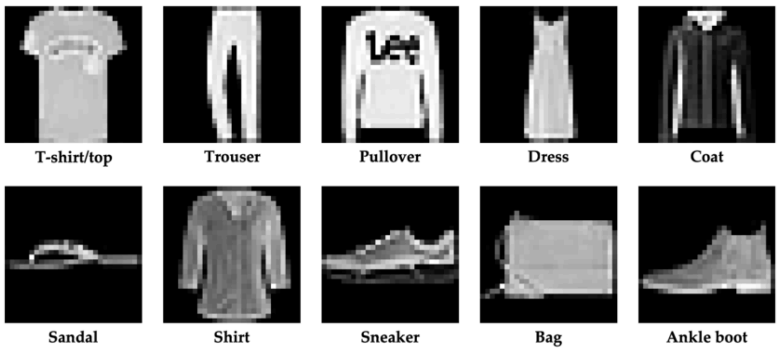
The goal of the lab is to implement the neural network for image classification and evaluate how various components and hyperparameters of a neural network and the training process affect the network's performance.

We have to use the FashionMNIST dataset and evaluate at least 3 values of:

- Learning rate,
- Mini-batch size (including a batch containing only 1 example),
- Number of hidden layers (including 0 hidden layers - a linear model),
- Width (number of neurons in hidden layers),
- Activation functions (e.g., Sigmoid, ReLU, GELU).

Dataset

The dataset used in the task is a FashionMNIST dataset, which consists of various pictures of clothes with the type assigned:

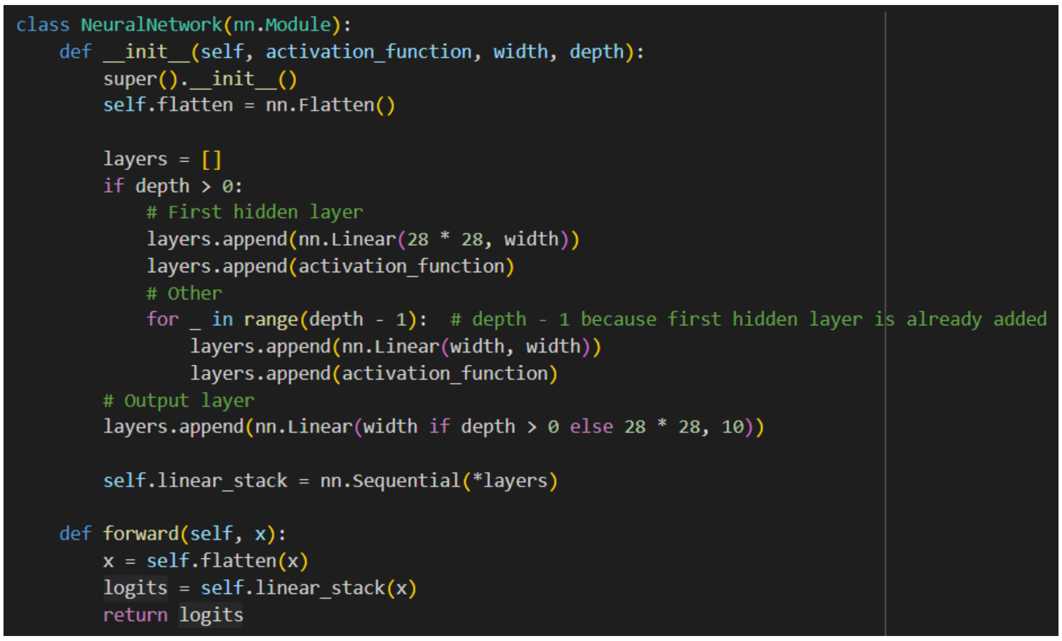


The dataset contains 70,000 grayscale images in 10 categories. The images show individual articles of clothing at low resolution (28 by 28 pixels). Here, 60,000 images are used to train the network, and 10,000 images to evaluate how accurately the network learned to classify images.

The data must be preprocessed before training the network. ToTensor converts a PIL image or NumPy ndarray into a FloatTensor. and scales the image's pixel intensity values in the range [0, 1] - exactly what we need.

Model

We are creating a model based on pytorch tutorial:



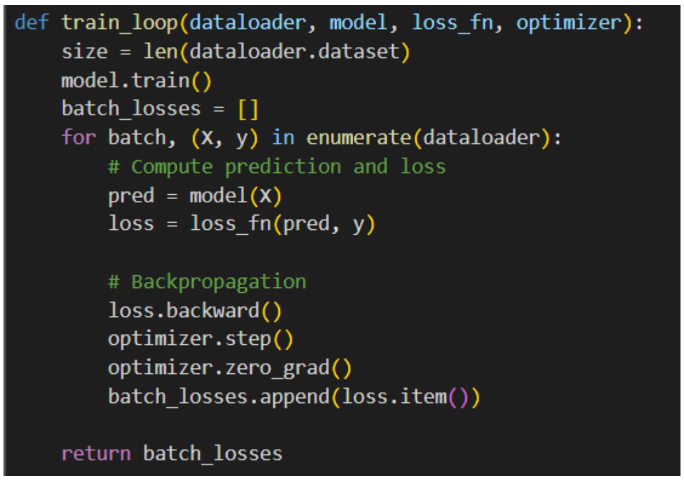
```python
class NeuralNetwork(nn.Module):
    def __init__(self, activation_function, width, depth):
        super().__init__()
        self.flatten = nn.Flatten()

        layers = []
        if depth > 0:
            # First hidden layer
            layers.append(nn.Linear(28 * 28, width))
            layers.append(activation_function)
            # Other
            for _ in range(depth - 1):  # depth - 1 because first hidden layer is already added
                layers.append(nn.Linear(width, width))
                layers.append(activation_function)
        # Output layer
        layers.append(nn.Linear(width if depth > 0 else 28 * 28, 10))

        self.linear_stack = nn.Sequential(*layers)

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_stack(x)
        return logits
```

First of all, there is a Flatten layer to convert each 2D 28x28 image into a contiguous array of 784 pixel values.

The model can have a different number of hidden layers, and the activation function can be set independently. The first hidden layer always accepts a 28*28 picture and performs a simple transformation using a weighted sum. Next, the data is passed to the activation function: it applies a non-linear transformation to the output of each neuron, allowing the network to learn and represent complex patterns. Without activation functions, the entire network behaves like a single linear function, no matter how deep.

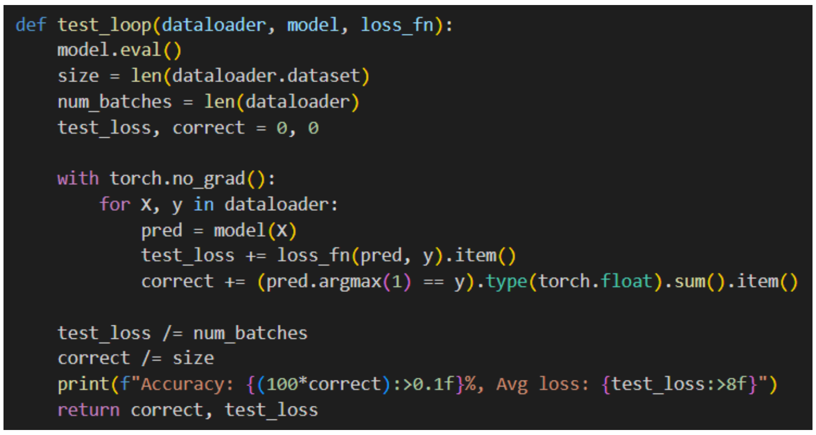
During each learning epoch, the model goes through two phases:

1. Training, where the model goes through the dataset batch by batch and tries to optimize the loss value:



```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    batch_losses = []
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        batch_losses.append(loss.item())

    return batch_losses
```

2. Testing, where the model is applied to a testing subset of data, which the model did not see during the training phase:



```
def test_loop(dataloader, model, loss_fn):
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f}")
    return correct, test_loss
```

Parameters

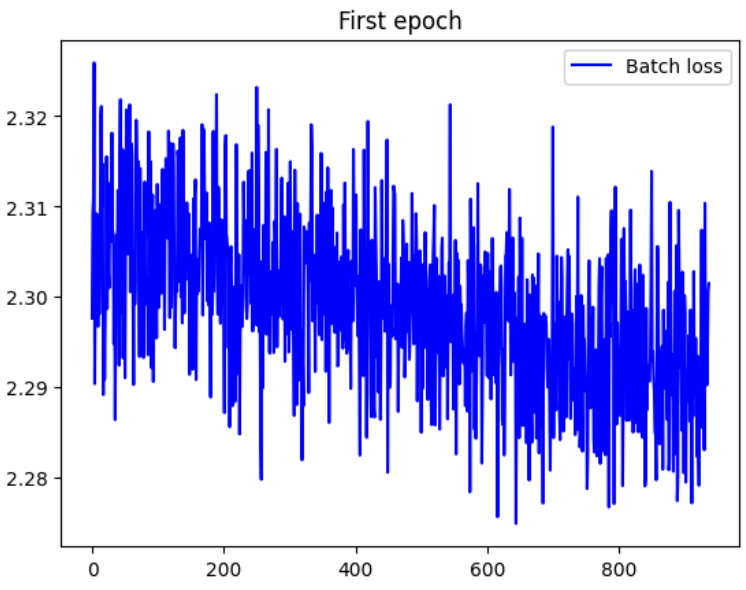
Here and later on, we are using the following common parameters:

- Loss function: CrossEntropyLoss
- optimizer: SGD

As a first experiment, we are using the default parameters from the tutorial:

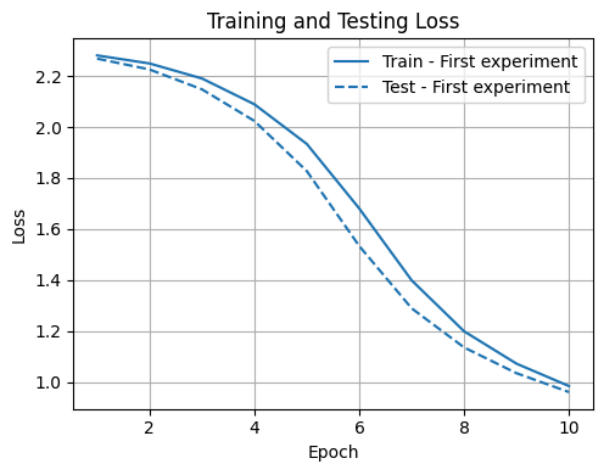
- batch_size = 64
- learning_rate = 1e-3
- depth = 3
- width = 128
- activation function: ReLU

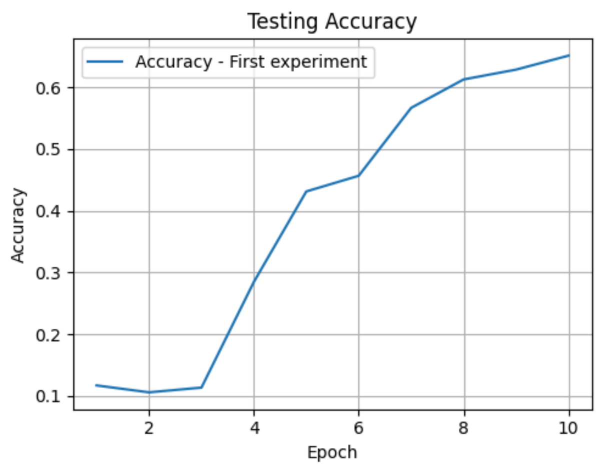
First of all, we can see how the neural network is learning during a single epoch by observing the loss value for each batch. The learning is not quite stable yet, but we can clearly see improvement batch over batch:



With these parameters, on 10 epochs, we observe the following results:

```
Epoch 1     Accuracy: 11.7%, Avg loss: 2.267708
Epoch 2     Accuracy: 10.6%, Avg loss: 2.225224
Epoch 3     Accuracy: 11.4%, Avg loss: 2.147393
Epoch 4     Accuracy: 28.5%, Avg loss: 2.023648
Epoch 5     Accuracy: 43.1%, Avg loss: 1.827293
Epoch 6     Accuracy: 45.7%, Avg loss: 1.533989
Epoch 7     Accuracy: 56.7%, Avg loss: 1.289878
Epoch 8     Accuracy: 61.3%, Avg loss: 1.136752
Epoch 9     Accuracy: 62.8%, Avg loss: 1.036010
Epoch 10    Accuracy: 65.1%, Avg loss: 0.962035
```





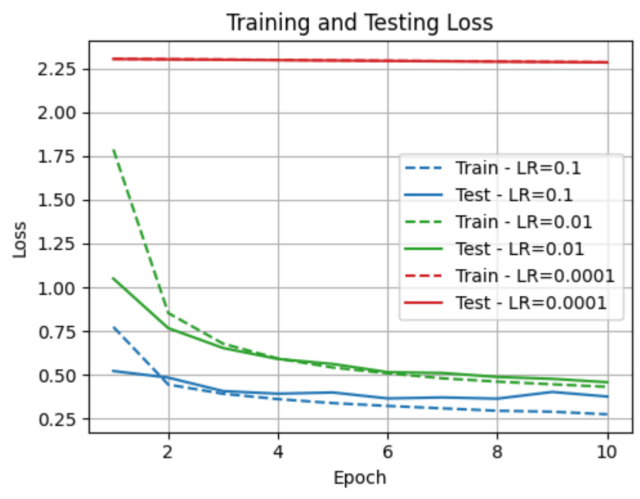
Learning rate

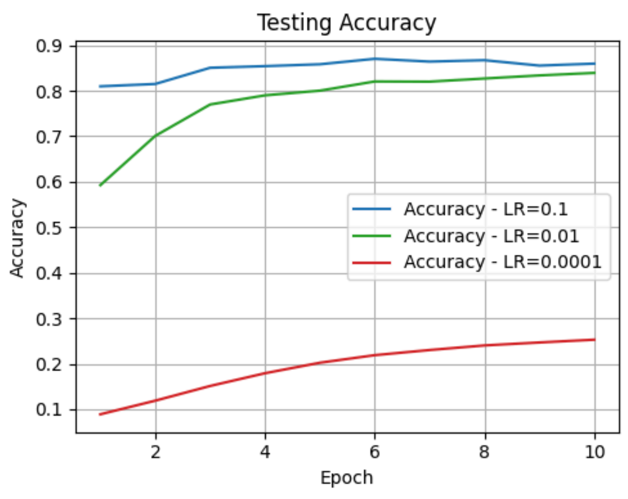
The learning rate controls how quickly the model updates its weights during training. While leaving all other parameters untouched, we try to run the model with the following values of learning rate:

- **0.0001** (very low, slow learning),
- **0.01** (commonly used default),
- **0.1** (aggressive, faster updates but risk of overshooting minima).

Takeaways:

- Optimal Learning Rate: 0.1 gives the best balance of speed and accuracy, though 0.01 is more stable and nearly as good.
- We haven't tried too High Learning Rate, because it could lead to divergence or instability.
- Too Low Learning Rate: at 0.0001, the model can't escape the initial state. It converges too slowly or not at all.





Batch size

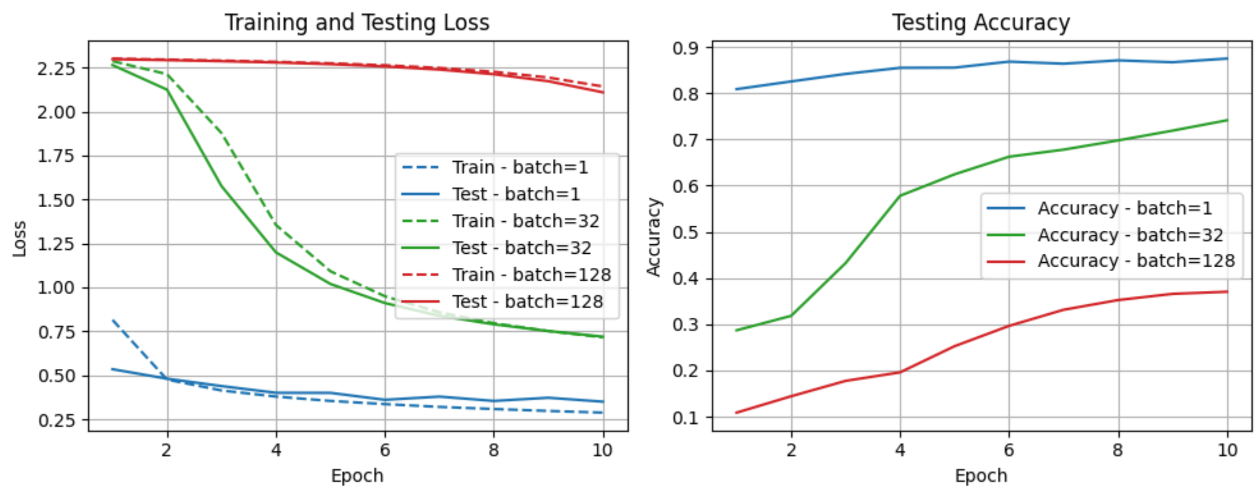
Batch size determines how many samples are processed before the model updates. Crucial parameter for the optimizer (SGD). Smaller sizes offer more frequent updates, while larger batches provide smoother gradients. While SGD usually refers to a batch size of 1, the SGD optimizer in PyTorch can still be used with mini-batches.

We experimented with:

- **1** (pure SGD),
- **32** (a typical size for mini-batch GD),
- **128** (large batch size for efficiency).

Takeaways:

- Having a batch size of 1 results in extremely long learning, as the model evaluates each picture individually. In our case, learning takes about 15 minutes. Works well for quick initial improvements, but slows down as the model stabilizes.
- Mini-batch size = 32 seems to be the sweet spot in this experiment: it provides a good balance between stability and fast convergence.
- Batch size = 128 leads to very slow convergence, with minimal improvement, suggesting that this batch size is not suitable.

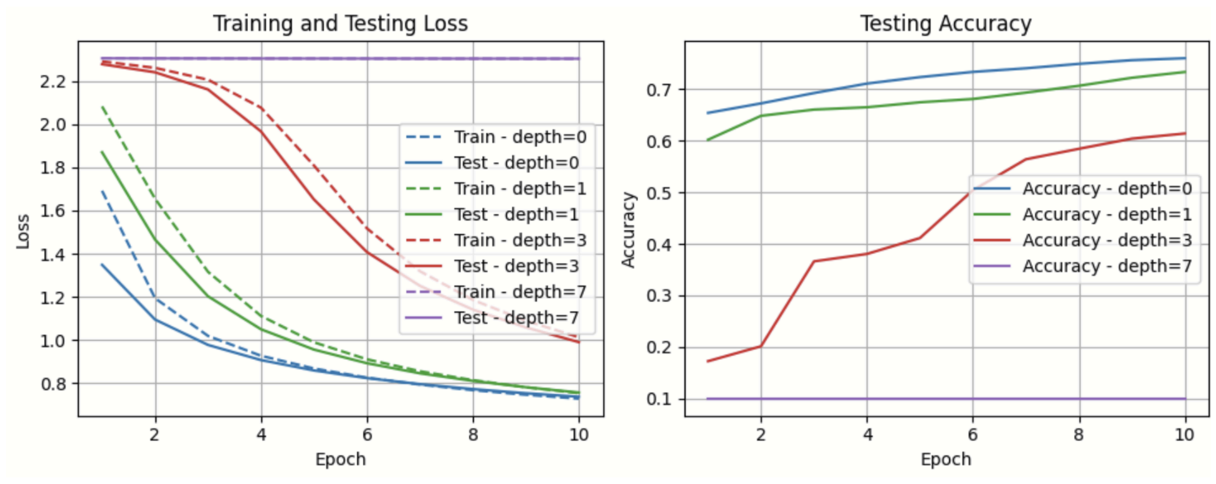


Hidden layers

The depth of the network affects its capacity to learn complex representations. We evaluated:

- **0 hidden layers** (linear transformation),
- **1 hidden layer,**
- **3 hidden layers,**
- **7 hidden layers** (overfitting risk).

with a constant width of 128 for each hidden layer



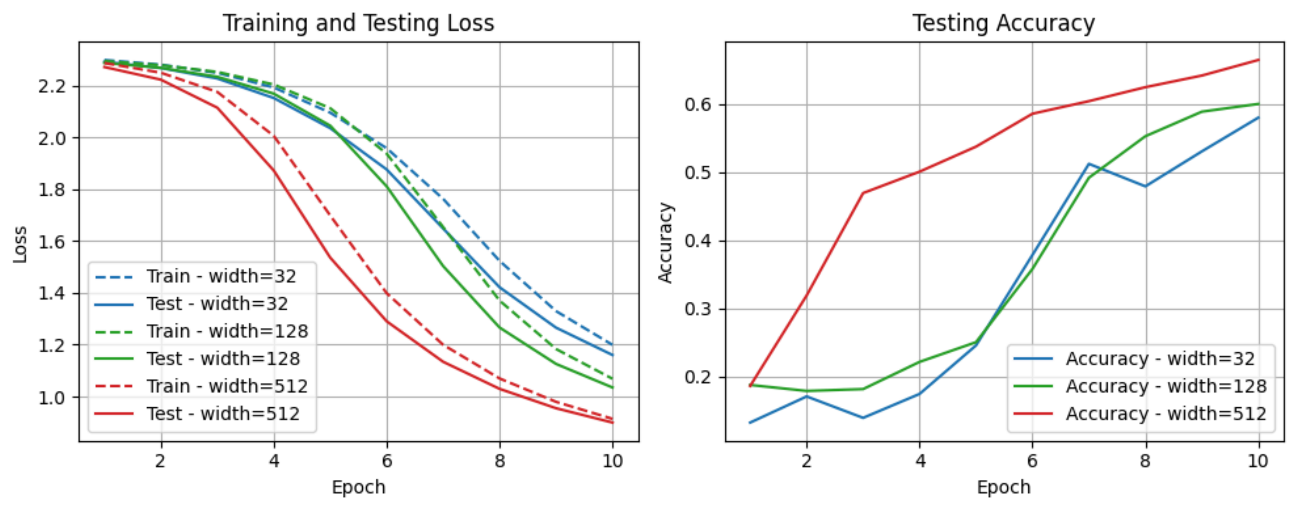
We can observe depth of 7 clearly overfitting.

Interestingly, depth of 1 showed best accuracy, while being stable - we observe no underfitting. Indicating that the data is very close to linearly separable.

Width

Width refers to the number of neurons in each hidden layer. Wider layers can model more complex functions but may also overfit. We tried:

- **32** neurons (narrow),
- **128** neurons (moderate),
- **512** neurons (wide).



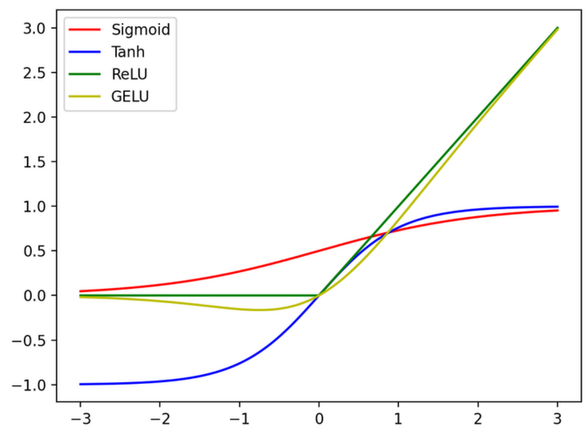
We can see that the width of 512 performed best.

The higher the width, the more stable the accuracy is.
Interestingly, the width of 32 performed similarly to the width of 512.

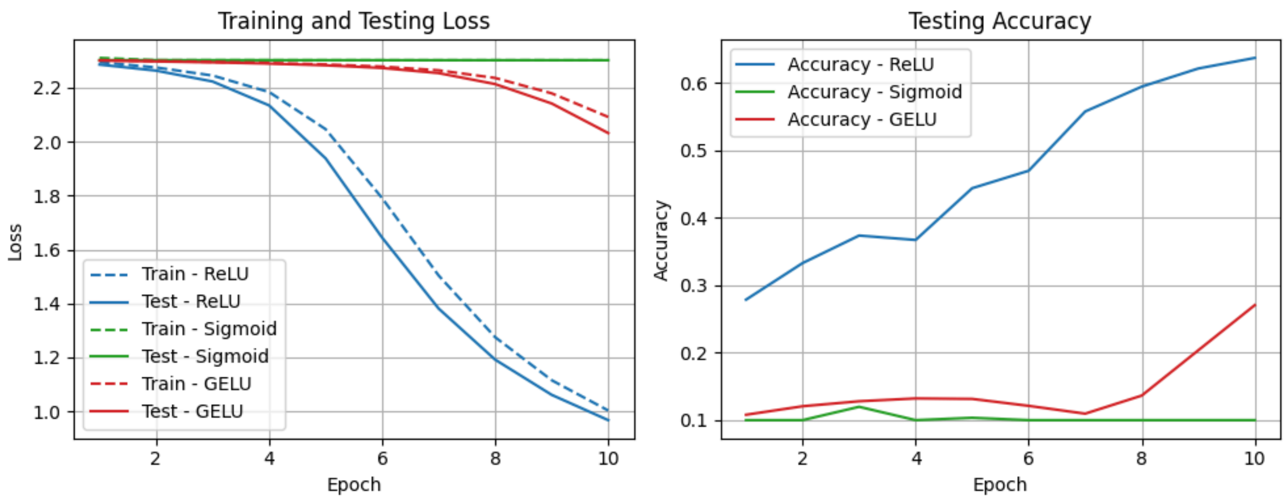
Activation functions

Activation functions introduce non-linearity and affect gradient flow. We compared:

- **Sigmoid** (classic but prone to vanishing gradients),
- **ReLU** (commonly used, efficient),
- **GELU** (more advanced, smooth approximation with modern advantages).



ReLU outperformed Sigmoid and GELU due to its simplicity and effectiveness in avoiding vanishing gradients.



Searching for hyperparameters

To avoid running all possible combinations of parameters, we decided to use optuna to find the best combination. Optuna is an automatic hyperparameter optimization software framework, particularly designed for machine learning.

We are testing parameters:

- learning rate in [1e-4, 1e-2, 1e-1]
- batch size between 1 and 128
- depth between 0 and 6
- width in [32, 128, 512]
- activation function in ["ReLU", "Sigmoid", "GELU"]

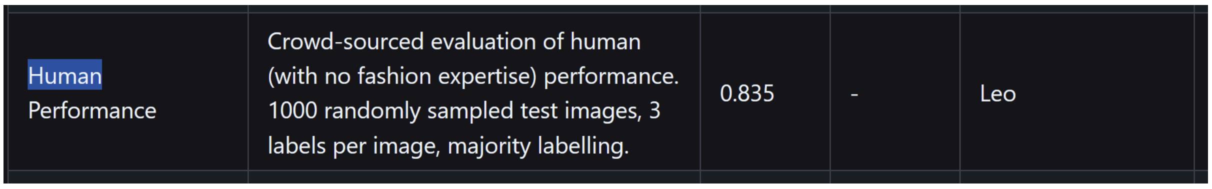
Results:

```
[I 2025-05-03 12:23:22,244] Trial 0 finished with value: 0.8136 and parameters:
{'lr': 0.01, 'batch_size': 127, 'depth': 0, 'width': 128, 'activation_fn': 'ReLU'}.
Best is trial 0 with value: 0.8136.

Accuracy: 81.4%, Avg loss: 0.556224

[I 2025-05-03 12:24:45,552] Trial 1 finished with value: 0.8607 and parameters:
{'lr': 0.1, 'batch_size': 41, 'depth': 3, 'width': 128, 'activation_fn': 'ReLU'}.
Best is trial 1 with value: 0.8607.

Accuracy: 86.1%, Avg loss: 0.415715

[I 2025-05-03 12:25:56,867] Trial 2 finished with value: 0.1 and parameters: {'lr':
0.01, 'batch_size': 104, 'depth': 6, 'width': 128, 'activation_fn': 'Sigmoid'}.
Best is trial 1 with value: 0.8607.

Accuracy: 10.0%, Avg loss: 2.303710

[I 2025-05-03 12:27:53,824] Trial 3 finished with value: 0.1 and parameters: {'lr':
0.0001, 'batch_size': 35, 'depth': 3, 'width': 512, 'activation_fn': 'Sigmoid'}.
Best is trial 1 with value: 0.8607.

Accuracy: 10.0%, Avg loss: 2.302418

[I 2025-05-03 12:29:02,519] Trial 4 finished with value: 0.7826 and parameters:
{'lr': 0.01, 'batch_size': 65, 'depth': 1, 'width': 128, 'activation_fn':
'Sigmoid'}. Best is trial 1 with value: 0.8607.

Accuracy: 78.3%, Avg loss: 0.607853

[I 2025-05-03 12:30:26,943] Trial 5 finished with value: 0.1 and parameters: {'lr':
0.0001, 'batch_size': 33, 'depth': 6, 'width': 32, 'activation_fn': 'ReLU'}. Best
is trial 1 with value: 0.8607.

Accuracy: 10.0%, Avg loss: 2.307202

[I 2025-05-03 12:31:28,662] Trial 6 finished with value: 0.8356 and parameters:
{'lr': 0.1, 'batch_size': 72, 'depth': 0, 'width': 128, 'activation_fn': 'ReLU'}.
Best is trial 1 with value: 0.8607.

Accuracy: 83.6%, Avg loss: 0.462638

[I 2025-05-03 13:04:41,482] Trial 7 finished with value: 0.1 and parameters: {'lr':
0.0001, 'batch_size': 1, 'depth': 6, 'width': 512, 'activation_fn': 'Sigmoid'}.
Best is trial 1 with value: 0.8607.

Accuracy: 10.0%, Avg loss: 2.306407

[I 2025-05-03 13:05:46,207] Trial 8 finished with value: 0.7577 and parameters:
{'lr': 0.01, 'batch_size': 78, 'depth': 2, 'width': 32, 'activation_fn': 'ReLU'}.
Best is trial 1 with value: 0.8607.

Accuracy: 75.8%, Avg loss: 0.621315

[I 2025-05-03 13:06:49,282] Trial 9 finished with value: 0.8441 and parameters:
{'lr': 0.1, 'batch_size': 76, 'depth': 1, 'width': 32, 'activation_fn': 'Sigmoid'}.
Best is trial 1 with value: 0.8607.

Accuracy: 84.4%, Avg loss: 0.425480
```

```
Best trial:
'tAccuracy: 0.8607
      Params:
            lr: 0.1
            batch_size: 41
            depth: 3
            width: 128
            activation_fn: ReLU
```

Conclusion

The search for the hyperparameters shows that the combination of the parameters is what matters the most.
There are some patterns and default values when searching for the optimal parameters, but at the end of the day, it is about a large number of trials to find the best combination of parameters that will work best for the particular task.
Of course, we didn't try all the combinations, because it would have taken ages.
We were able to achieve an accuracy of 86%, which is better than human performance.



Human Performance	Crowd-sourced evaluation of human (with no fashion expertise) performance. 1000 randomly sampled test images, 3 labels per image, majority labelling.	0.835	-	Leo