Assignment 2

Deep Learning KU, WS 2025/26

Team Members		
Last name	First name	Matriculation Number
Bajkusa	Mateo	11943093
Jusic	Enis	12004830

Task 1

Task 1.1 - Data Inspection and Splitting

Splitting the dataset into three parts as we did, is a common way to train, validate and test a Neural Network (NN). The biggest dataset will be used for training, where the NN learns by updating its parameters. We include the validation set where we validate the network during training, to tweak hyper-parameters or detect overfitting. Finally, the Test set is used once in the end and provides an evaluation of the final model's performance.

Task 1.2 - Feature Visualization

For visualization we have decided on three plots: a) Histogram: from this plot we can see the frequency of different parameters over their respective values. This is important, because some of our parameters vary highly and therefore it is necessary to standardize the dataset.

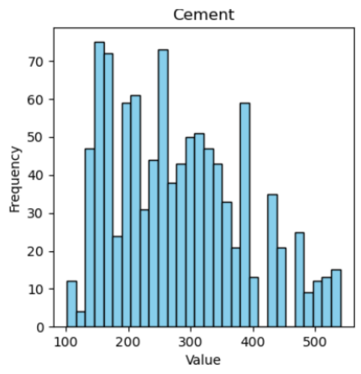


Figure 1: Frequency distribution of the Cement feature.

b) Box-Plot: This plot helps us to identify skewness and outliers. As shown in Figure 2, the 'Age' feature is highly right-skewed, meaning that most concrete is tested early (within 30 days), while the black circles on the right represent outliers (older concrete).

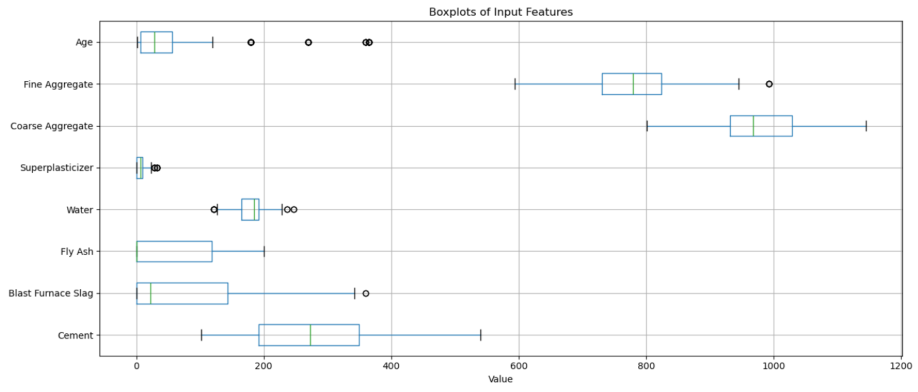


Figure 2: Boxplots of all input features showing outliers and skewness.

c) Histogram2D - Finally we have a relation between our target variable Strength and all the other features. Figure 3 shows a that higher concentrations of cement generally correlate with higher strength. The yellow areas indicate the most common combinations of cement and strength in our dataset

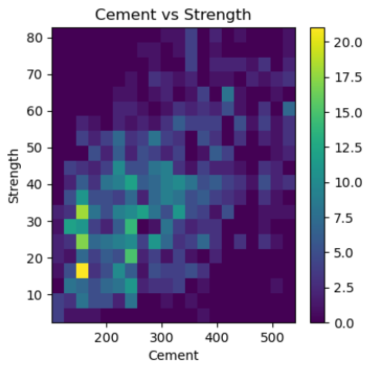


Figure 3: 2D Histogram showing the density of samples for Cement vs. Strength.

Task 1.3 - Feature Scaling

From our histogram we gathered the information that some features vary a lot. So we will standardize to prevent large features form dominating the learning process. Standardization will ensure all features contribute fairly to the learning process. And we only fit our training set as not to learn anything about our test or validation sets.

Task 1.4 - DataLoader Construction

First, we converted the standardized Numpy arrays into PyTorch Tensors and wrapped them in TensorDataset objects.

For the DataLoaders, we selected a batch size of 64. Given the dataset size (≈ 1000 samples), this size provides a good balance: it is small enough to ensure the model receives frequent weight updates, but large enough for fast training. We will at a later point test batch size 32 and 128 and compare the results.

We enabled shuffling for the training set to prevent the network from memorizing the sequence of the data, forcing it to learn the relationships between features and our target feature. The validation and test set are not shuffled, as the order does not affect evaluation metrics.

Task 2

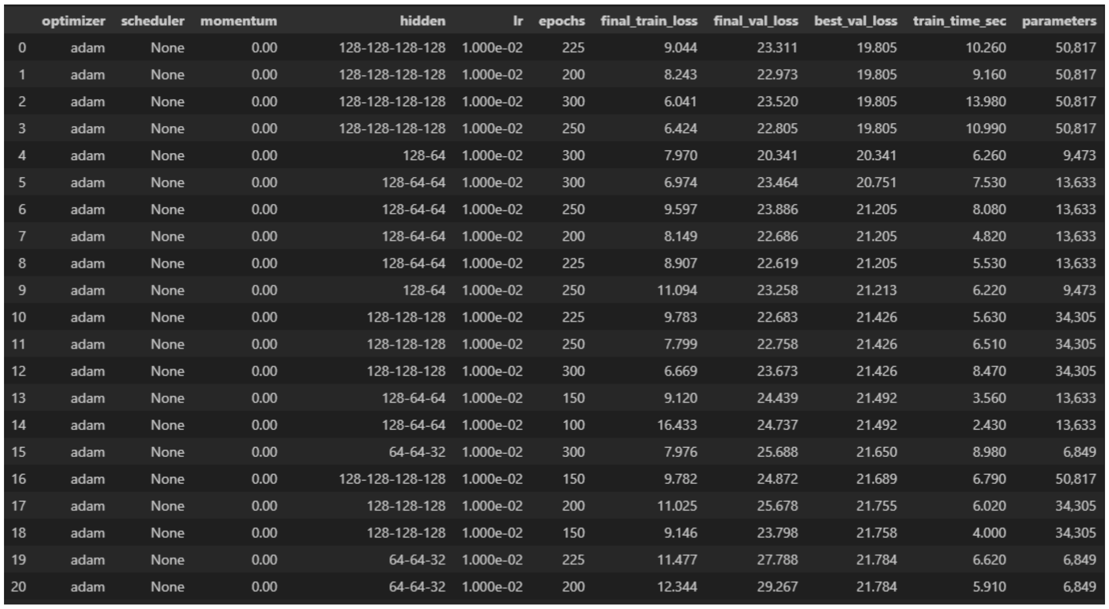
Task 2.1 - Model Design and Training Setup

To implement our NN, we used nn.Sequential to dynamically construct the network based on the list of hidden layers.

- **Hidden Layers:** We experimented we numerous hidden layer sizes for example: [32], [64,32], [12,12,12,12], [128,128,128] and many more. We used ReLU activation functions for all hidden layers as was specified in the assignment.
- **Output Layer:** The final layer is a single neuron with no activation function. We are dealing with a regression task of predicting a continuous target, and therefore we choose not to have an activation func.
- **Loss Function:** We used the Mean Squared Error (MSE) loss function, which penalizes large errors and is the standard for regression problems. We decided not to go with the MAE loss (L1 loss) as our data did not have huge outliers.
- **Training Configuration:**
  - **Optimizer:** We used ADAM as the baseline optimizer because its most common optimizer and provides a good start. In later tasks we test multiple optimizers to find the best one.
  - **Hyperparameters:** We selected a batch size of 64 and trained for $100, 200, 50, 150...$ epochs. Finally the learning rate, as we later do learning rate scheduling, currently we put the learning rate to 1e-2, as a start, beacuse it a fairly large rate and it will allow our models to coverge quickly.

Task 2.2 - Training and Results Comparison

In this task we have set the learning rate to 1e-2 and the optimizer to ADAM.



	optimizer	scheduler	momentum	hidden	lr	epochs	final_train_loss	final_val_loss	best_val_loss	train_time_sec	parameters
0	adam	None	0.00	128-128-128-128	1.000e-02	225	9.044	23.311	19.805	10.260	50,817
1	adam	None	0.00	128-128-128-128	1.000e-02	200	8.243	22.973	19.805	9.160	50,817
2	adam	None	0.00	128-128-128-128	1.000e-02	300	6.041	23.520	19.805	13.980	50,817
3	adam	None	0.00	128-128-128-128	1.000e-02	250	6.424	22.805	19.805	10.990	50,817
4	adam	None	0.00	128-64	1.000e-02	300	7.970	20.341	20.341	6.260	9,473
5	adam	None	0.00	128-64-64	1.000e-02	300	6.974	23.464	20.751	7.530	13,633
6	adam	None	0.00	128-64-64	1.000e-02	250	9.597	23.886	21.205	8.080	13,633
7	adam	None	0.00	128-64-64	1.000e-02	200	8.149	22.686	21.205	4.820	13,633
8	adam	None	0.00	128-64-64	1.000e-02	225	8.907	22.619	21.205	5.530	13,633
9	adam	None	0.00	128-64	1.000e-02	250	11.094	23.258	21.213	6.220	9,473
10	adam	None	0.00	128-128-128	1.000e-02	225	9.783	22.683	21.426	5.630	34,305
11	adam	None	0.00	128-128-128	1.000e-02	250	7.799	22.758	21.426	6.510	34,305
12	adam	None	0.00	128-128-128	1.000e-02	300	6.669	23.673	21.426	8.470	34,305
13	adam	None	0.00	128-64-64	1.000e-02	150	9.120	24.439	21.492	3.560	13,633
14	adam	None	0.00	128-64-64	1.000e-02	100	16.433	24.737	21.492	2.430	13,633
15	adam	None	0.00	64-64-32	1.000e-02	300	7.976	25.688	21.650	8.980	6,849
16	adam	None	0.00	128-128-128-128	1.000e-02	150	9.782	24.872	21.689	6.790	50,817
17	adam	None	0.00	128-128-128	1.000e-02	200	11.025	25.678	21.755	6.020	34,305
18	adam	None	0.00	128-128-128	1.000e-02	150	9.146	23.798	21.758	4.000	34,305
19	adam	None	0.00	64-64-32	1.000e-02	225	11.477	27.788	21.784	6.620	6,849
20	adam	None	0.00	64-64-32	1.000e-02	200	12.344	29.267	21.784	5.910	6,849

Figure 4: Table of results from various sizes for Hidden layers in NN

From figure 4 we can clearly see that the best architecture is [128, 128, 128, 128], it had the lowest validation score (19.805) and a low final training loss (9.044). First four rows with the same architecture all have the same validation loss. Although the larger architecture achieved the overall best score, we would select the 4th row architecture [128,64] as our top pic. It had a negligible difference in validation loss (19.805 vs 20.341) but it had a much lower computation time and significantly lower amount of parameters, and because of its size it gives us less chance to overfit.

Task 2.3 - Parameter Count

The total number of trainable parameters consists of the weights and biases for each fully connected layer. We verify the computation for the architecture **[128, 64]** (Input dim: 8).

- **Layer 1 ($8 \to 128$):** $(8 \times 128) + 128 = 1,152$ parameters.
- **Layer 2 ($128 \to 64$):** $(128 \times 64) + 64 = 8,256$ parameters.
- **Output Layer ($64 \to 1$):** $(64 \times 1) + 1 = 65$ parameters.

Total: $1,152 + 8,256 + 65 = \mathbf{9,473}$. This matches the value reported in the code.

Task 2.4 - Learning Curves

We selected three distinct models ([128, 128, 128, 128], [128, 64] and [128, 128, 128]) to analyze the training dynamics. Figure 5 displays the training and validation losses across epochs.

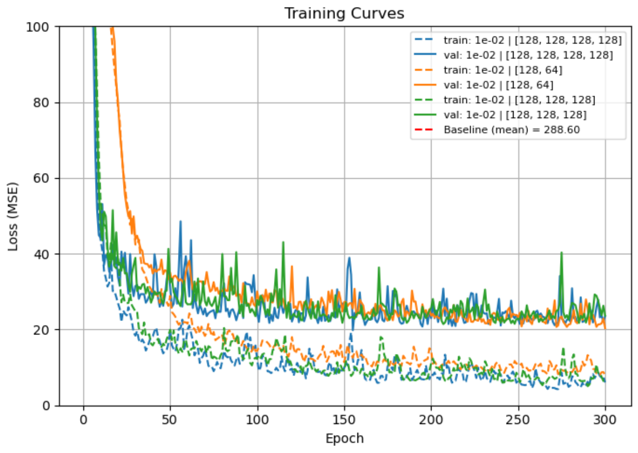


Figure 5: Training and validation loss (MSE) over 300 epochs. The Baseline is at 288.60, which is outside the frame.

The plots demonstrate fast convergence for all models, with the loss dropping significantly below the baseline within the first few epochs. We observe signs of overfitting, as the validation loss (solid line) diverges from the training loss (dashed line), while training loss continues to get lower.

Task 3

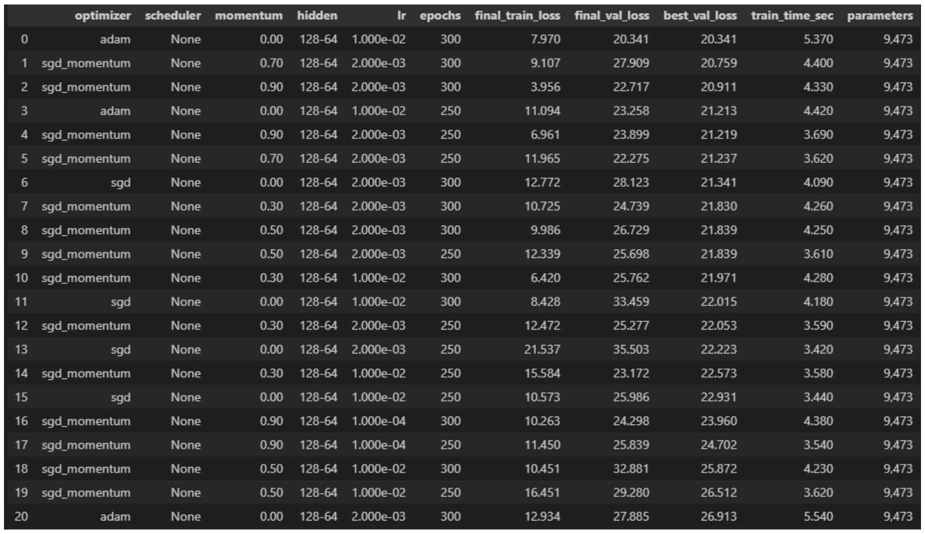
Task 3.1 - Optimizer and Training Setup

The best chosen model is [128, 64] because of its validation loss, short training time, and low number of parameters.

For the optimizer comparison, we evaluated three configurations:

1. **SGD:** Standard Stochastic Gradient Descent.
2. **SGD with Momentum:** SGD with a momentum factor raging from 0.5 to 0.9.
3. **Adam:** Adaptive Moment Estimation.

Initially, we attempted a learning rate of 1e-2 for all optimizers. While Adam converged successfully, both SGD variants resulted in NaN loss due to exploding gradients. This occurred because the target variable (Strength) is unscaled, leading to large MSE gradients that destabilized SGD. Consequently, we tried decreasing the learning rate, which helped SDG optimizers but made ADAM very bad. To resolve the problem we tested the implementation with a couple of variants of learning rates. Epochs were choosing based on the best results form figure 4 which were 250 and 300.



	optimizer	scheduler	momentum	hidden	lr	epochs	final_train_loss	final_val_loss	best_val_loss	train_time_sec	parameters
0	adam	None	0.00	128-64	1.000e-02	300	7.970	20.341	20.341	5.370	9,473
1	sgd_momentum	None	0.70	128-64	2.000e-03	300	9.107	27.909	20.759	4.400	9,473
2	sgd_momentum	None	0.90	128-64	2.000e-03	300	3.956	22.717	20.911	4.330	9,473
3	adam	None	0.00	128-64	1.000e-02	250	11.094	23.258	21.213	4.420	9,473
4	sgd_momentum	None	0.90	128-64	2.000e-03	250	6.961	23.899	21.219	3.690	9,473
5	sgd_momentum	None	0.70	128-64	2.000e-03	250	11.965	22.275	21.237	3.620	9,473
6	sgd	None	0.00	128-64	2.000e-03	300	12.772	28.123	21.341	4.090	9,473
7	sgd_momentum	None	0.30	128-64	2.000e-03	300	10.725	24.739	21.830	4.260	9,473
8	sgd_momentum	None	0.50	128-64	2.000e-03	300	9.986	26.729	21.839	4.250	9,473
9	sgd_momentum	None	0.50	128-64	2.000e-03	250	12.339	25.698	21.839	3.610	9,473
10	sgd_momentum	None	0.30	128-64	1.000e-02	300	6.420	25.762	21.971	4.280	9,473
11	sgd	None	0.00	128-64	1.000e-02	300	8.428	33.459	22.015	4.180	9,473
12	sgd_momentum	None	0.30	128-64	2.000e-03	250	12.472	25.277	22.053	3.590	9,473
13	sgd	None	0.00	128-64	2.000e-03	250	21.537	35.503	22.223	3.420	9,473
14	sgd_momentum	None	0.30	128-64	1.000e-02	250	15.584	23.172	22.573	3.580	9,473
15	sgd	None	0.00	128-64	1.000e-02	250	10.573	25.986	22.931	3.440	9,473
16	sgd_momentum	None	0.90	128-64	1.000e-04	300	10.263	24.298	23.960	4.380	9,473
17	sgd_momentum	None	0.90	128-64	1.000e-04	250	11.450	25.839	24.702	3.540	9,473
18	sgd_momentum	None	0.50	128-64	1.000e-02	300	10.451	32.881	25.872	4.230	9,473
19	sgd_momentum	None	0.50	128-64	1.000e-02	250	16.451	29.280	26.512	3.620	9,473
20	adam	None	0.00	128-64	2.000e-03	300	12.934	27.885	26.913	5.540	9,473

Figure 6: Table with results of training for different optimizers

The overall best results were form the ADAM optimizer with validation loss being 20.341. Close behind was sgd_momentum, at 2nd and 3rd place and with a slightly lower training time. But with training loss of 3.9 for sdg momentum of 0.9, we can be sure that this model is highly overfitting. And finally SDG has a higher val loss but also a significantly higher training loss, which could indicate slower convergence. But because of its place in the table, it should not a valid option. Regarding momentum, we had four different options: 0.9, 0.7,0.5 and 0.3. The viable momentum was either 0.9 or 0.7, the other two models had validation loss too high.

Task 3.2 - Learning Curves and Discussion

Figure 7 shows the plot results for different optimizers.

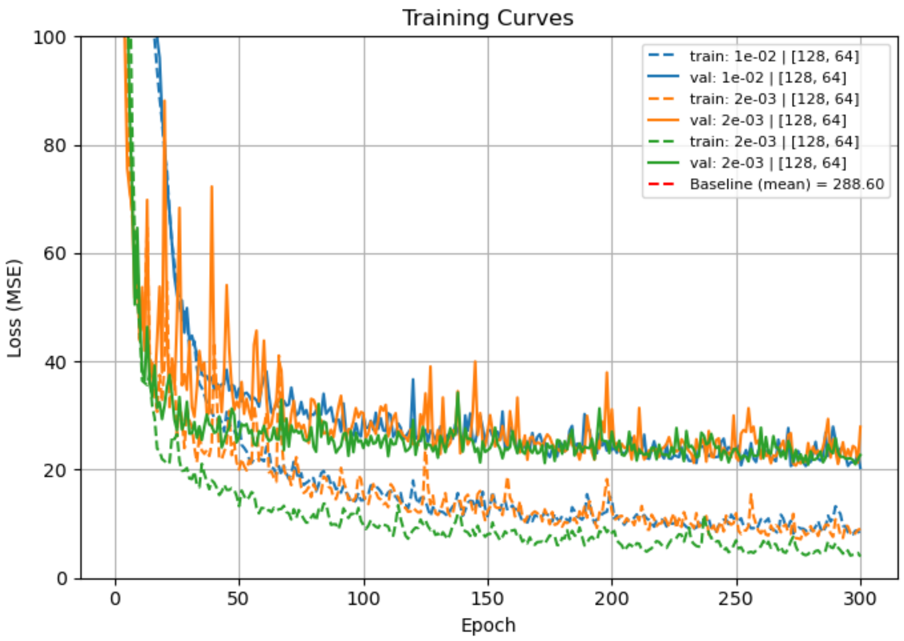


Figure 7: Plot with results of training for different optimizers without SDG

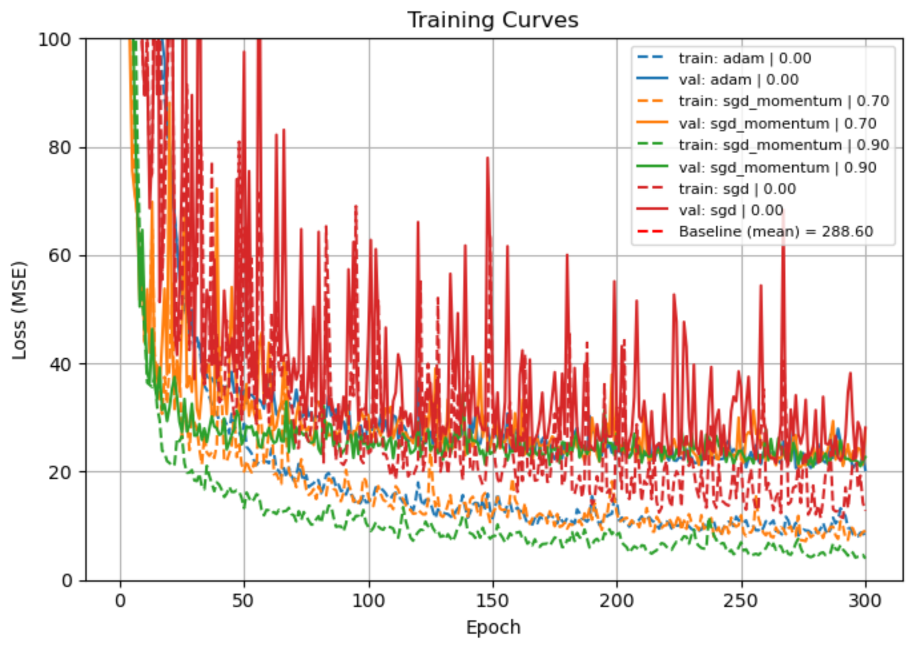


Figure 8: Plot with results of training for different optimizers including SDG

SGD optimizer makes the plot almost unreadable, so we provided two figures: one figure **??** without SGD and one figure 8 with SGD.

- **Convergence Speed:** SGD with high momentum ($\mu = 0.9$, green) demonstrates the most rapid convergence, dropping strictly in the first 10-20 epochs. Adam (blue) follows closely with a similarly fast descent. In strong contrast, standard SGD (red) shows the slowest convergence rate. Without the acceleration provided by momentum, the optimizer struggles to converge and is very erratic.
- **Stability:** The graph highlights a huge difference in stability. Standard SGD (red) exhibits extreme instability, characterized by massive high-frequency oscillations and spikes in loss throughout the entire training duration. This visualizes the noisy nature of gradient updates when not smoothed by past steps. Adding momentum significantly dampens this noise: $\mu = 0.7$ (orange) and $\mu = 0.9$ (green) reduce the variance, Adam achieves a middle ground, maintaining stability without the erratic behavior of standard SGD.
- **Generalization:** A trade-off is evident between training fit and generalization. SGD with $\mu = 0.9$ achieves the lowest training loss (≈ 4) but suffers from severe overfitting, as seen by the large gap between its training (dashed) and validation (solid) curves. Standard SGD (red) sits at

the opposite end of the spectrum; its higher final training loss suggests it failed to fully converge (underfitting) due to its inability to settle into a minimum effectively. ADAM provides an excellent result and a middle ground, as the validation and training loss are not far from each other.

- **Best Setup:** We pick ADAM as the best optimizer, as it provides the most stable configuration and convergence. SGD is too unstable and SGD with momentum is prone to overfitting.

Task 3.3 - Learning Rate Scheduling

Verifying the schedulers:

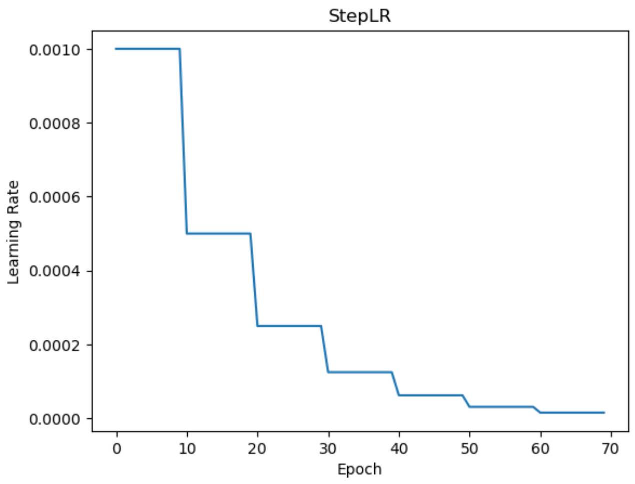
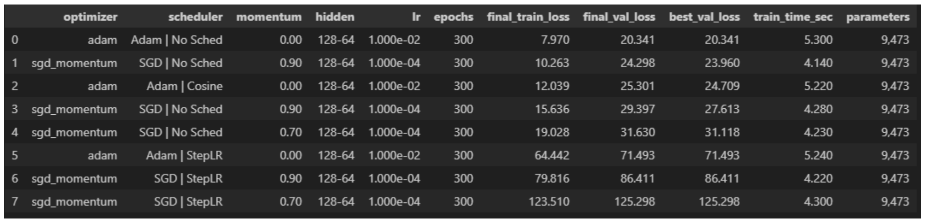


Figure 9: Plot of the StepLR scheduler

In the code we specified the step_size to be 10 and gamma to be 0.5. In the plot 9 we can see the drop in the learning rate each 10 epochs or steps by the size of 0.5. This versify that the schedulers are configured correctly.



	optimizer	scheduler	momentum	hidden	lr	epochs	final_train_loss	final_val_loss	best_val_loss	train_time_sec	parameters
0	adam	Adam | No Sched	0.00	128-64	1.000e-02	300	7.970	20.341	20.341	5.300	9,473
1	sgd_momentum	SGD | No Sched	0.90	128-64	1.000e-04	300	10.263	24.298	23.960	4.140	9,473
2	adam	Adam | Cosine	0.00	128-64	1.000e-02	300	12.039	25.301	24.709	5.220	9,473
3	sgd_momentum	SGD | No Sched	0.90	128-64	1.000e-04	300	15.636	29.397	27.613	4.280	9,473
4	sgd_momentum	SGD | No Sched	0.70	128-64	1.000e-04	300	19.028	31.630	31.118	4.230	9,473
5	adam	Adam | StepLR	0.00	128-64	1.000e-02	300	64.442	71.493	71.493	5.240	9,473
6	sgd_momentum	SGD | StepLR	0.90	128-64	1.000e-04	300	79.816	86.411	86.411	4.220	9,473
7	sgd_momentum	SGD | StepLR	0.70	128-64	1.000e-04	300	123.510	125.298	125.298	4.300	9,473

Figure 10: Table with the results of different schedulers

From figure 10 we have determined that scheduling does not greatly impact the final validation performance. SDG momentum and ADAM (no sched) are still the best optimizer.

The StepLR scheduler had the highest loss because it likely lowered the learning rate too quickly, stopping the model from learning before it was ready. The Cosine scheduler was stable but did not match the no scheduler variant.

For this specific dataset, using a constant learning rate is sufficient. Scheduling adds complexity but does not provide a significant benefit to convergence speed or final accuracy. The best overall model is still ADAM.

Task 4

Task 4.1 - Setup and Initialization

Chosen Configuration: Based on the experiments in Tasks 2 and 3, we selected the following configuration for weight investigation:

- **Architecture:** [128, 64].
- **Optimizer:** ADAM.
- **Learning Rate:** 1e-2.
- **Scheduler:** None.
- **Epochs:** 300.

We chose this architecture because it achieved low validation loss in Task 2 and 3 without overfitting. We selected ADAM with a learning rate of 1e-2 because it demonstrated a fast convergence and highest stability in Task 3.2. Finally, from task 3.3 we have determined that using a scheduler brings no benefit so we have decided not to use one.

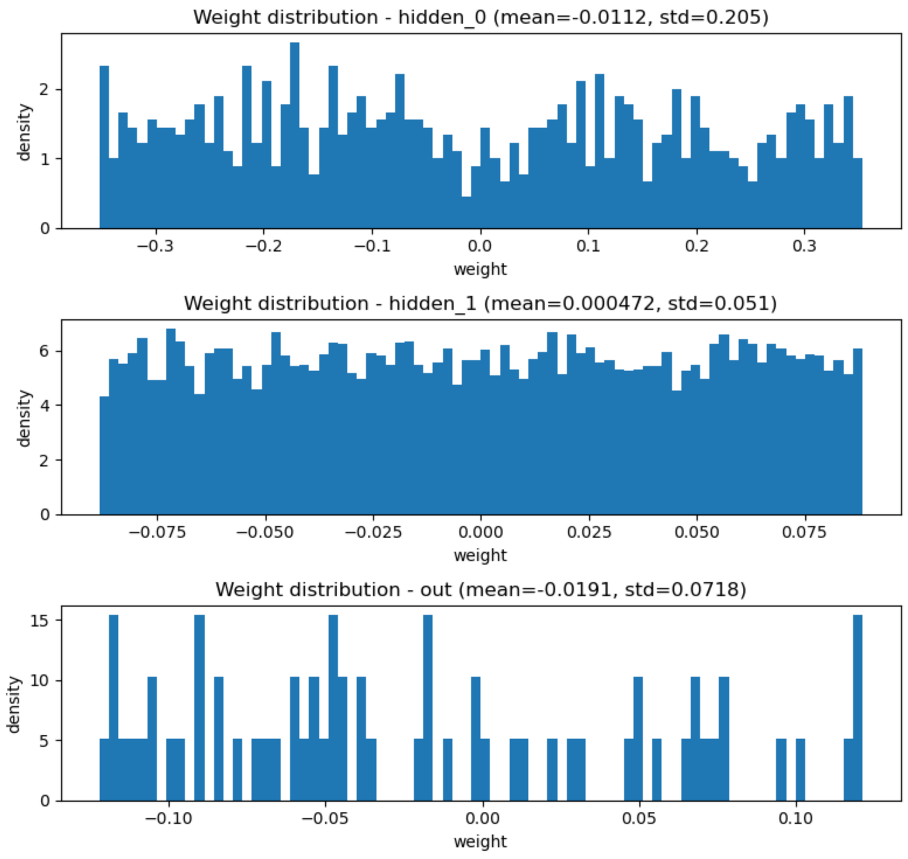


Figure 11: weight histogram representing an untrained model

We plotted the weight distributions of this model before training (see Figure 11). The histograms show a uniform distribution. According to the PyTorch documentation[1], linear layers are initialized using Kaiming Uniform, where weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ with $k = \frac{1}{\text{in\_features}}$. The plots verify this, showing rectangular distributions with bounds that scale down as the layer size increases.

[1] `https://pytorch.org/docs/stable/generated/torch.nn.Linear.html`

Task 4.2 - Trained Model Weights

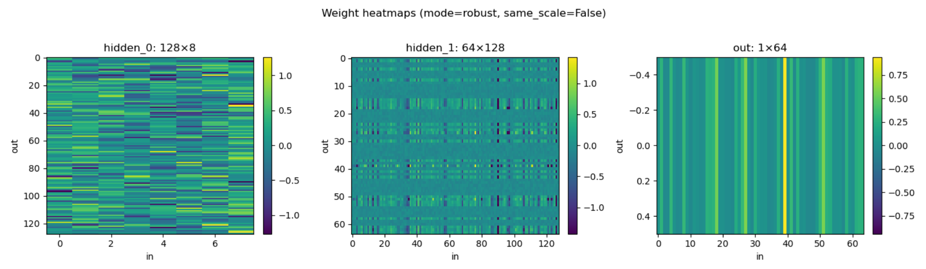


Figure 12: Heatmap representing the weight training of the best overall model.

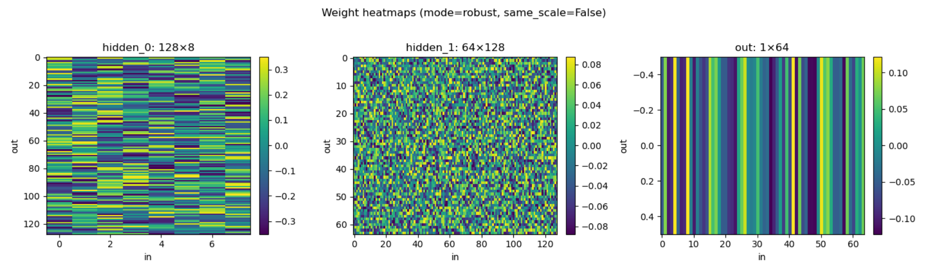


Figure 13: weight heatmap representing an untrained model

- **Layer 1 (hidden_0:** The first heatmap (128×8) shows the weights connecting the 8 input features to the 128 neurons of the first hidden layer. The weights appear dense and distributed; there is no obvious sparsity pattern (i.e., large sections of zero weights). This indicates that the network utilizes a combination of all input features to construct its initial hidden representation. If we also compare it to the network that is not trained, we can see that that one is not so reluctant to give really high or really low scores to weights.
- **Layer 2 (hidden_1):** In figure 13 the weights are completely random and resemble static noise. In figure 12 the network assigns mid range (0) values to all weights. During the training the Adam optimizer reduced the effects od random weights.
- **Output Layer (out):** The final heatmap (1×64) connects the second hidden layer to the single output neuron. Distinct vertical bands are visible, specifically a prominent bright yellow band around index 40. This suggests that the final prediction relies heavily on specific high-level features learned by the previous layer, while effectively ignoring others (represented by the darker, near-zero regions). On the untrained network we have a couple of prominent bright yellow bands.

Interpretability of Input Variables: We cannot observe influence of individual input variables on the target. Neural network mixes features trough non-linear activation functions and matrix multiplications. A big weight in the first layer does not mean that this input will have big effect on the output. And most importantly there is no way to follow that weight trough the network by just inspecting the heatmap.

Task 5

Task 5.1 – Gradients for the Current Setup

For the gradient investigation we used the best-performing setup from the task 3 which is a network with hidden layers $[128, 64]$, the adam optimizer with learning rate $\eta = 10^{-2}$, batch size 64 and 300 epochs.

During training we computed the mean absolute gradient for every parameter in every batch and averaged these values for the final epoch, following the definition in the assignment sheet

$$\bar{g}_\ell^{(e)} = \frac{1}{N_e} \sum_{i=1}^{N_e} \frac{1}{n_\ell} \frac{1}{|\Theta_{\ell,j}|} \sum_{\theta \in \Theta_{\ell,j}} |\nabla_\theta \mathcal{L}(x_i, y_i)| \, ,$$

where N_e is the number of samples seen in epoch e, n_ℓ is the number of neurons in layer ℓ, and $\Theta_{\ell,j}$ is the set of parameters of neuron j in layer ℓ.

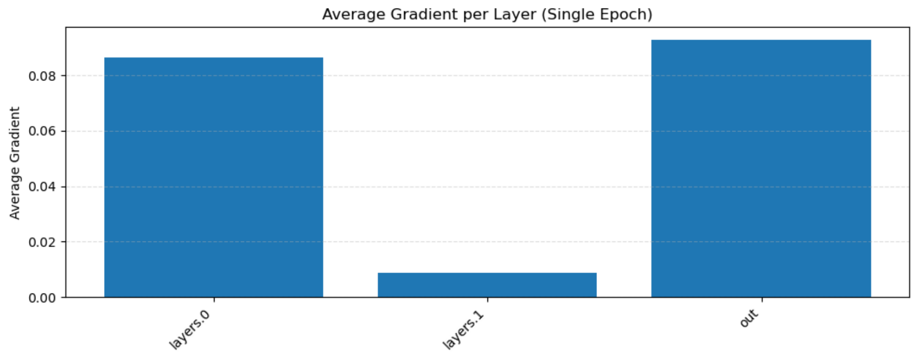


Figure 14: Average gradient per layer for the last training epoch

The resulting bar plot in Figure 14 shows that the output layer and the first hidden layer receive the largest gradient magnitudes, while the second hidden layer receives smaller gradients. This is normal behaviour for our 2-layer network that we have chosen because gradients shrink as they propagate backwards through the network in the backpropagation step. This indicates that for our chosen architecture and learning rate, the network does not suffer from vanishing or exploding gradient otherwise we would have seen gradients at 0.

Task 5.2 – Increasing Depth and Vanishing Gradients

To study the effect of depth and vanishing gradients, we fixed a small hidden width of 32 neurons per layer and exponentially increased the number of hidden layers for the setup $\{1, 2, 4, 8, 16, 32, 64, 128\}$. For each configuration we trained the model for 300 epochs with learning rate $\eta = 10^{-2}$ and adam as our optimizer and computed the average gradient magnitude per layer in the final epoch using the same procedure as in Task 5.1.

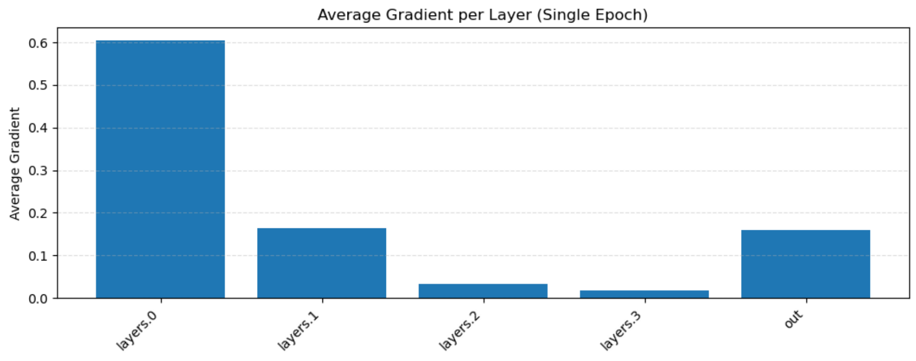


Figure 15: Average gradient per layer for a network of depth 4 hidden layers

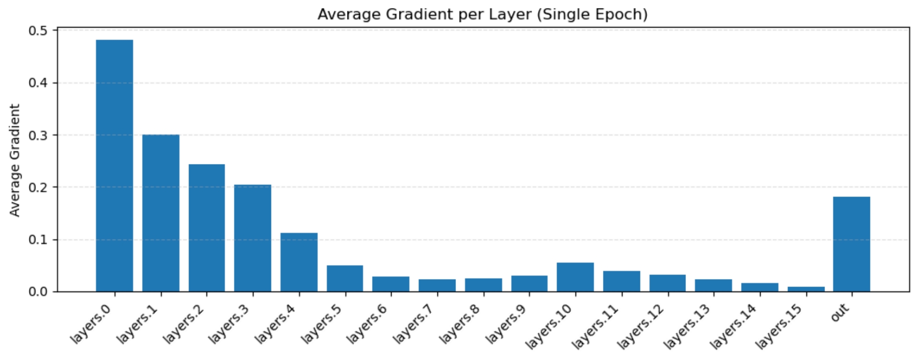


Figure 16: Average gradient per layer for a network of depth 16 hidden layers

Figure 15 and Figure 16 show the average gradient magnitude of all hidden layers and output layer in a 4 and 16 hidden layer network and indicate that for shallow networks the gradients remain relatively uniform across layers but the gradients in the layers close to the output layer shrink by several orders of magnitude compared to the input layer. This is already a small sign of vanishing gradients, but still nothing to really ”worry” about here.

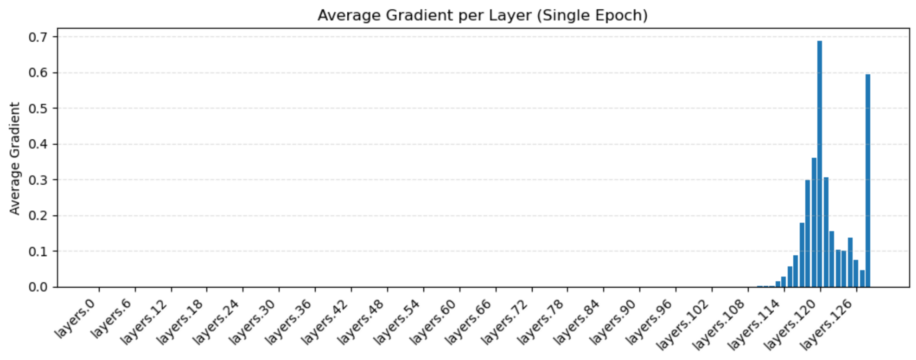


Figure 17: Average gradient per layer for a network with 128 hidden layers. The first 100 layers receive gradients close to zero, clearly showing vanishing gradients.

For deeper networks, like our 128 hidden layer network shown in Figure 17, the vanishing gradient problem becomes clearly visible. Only the last few layers near the output receive gradients of important magnitude, while the first hundred layers receive gradients that are zero or almost zero. These layers do not update during training and also do not contribute to the learning process. As a result the network behaves like a shallow model and loses the training effectiveness.

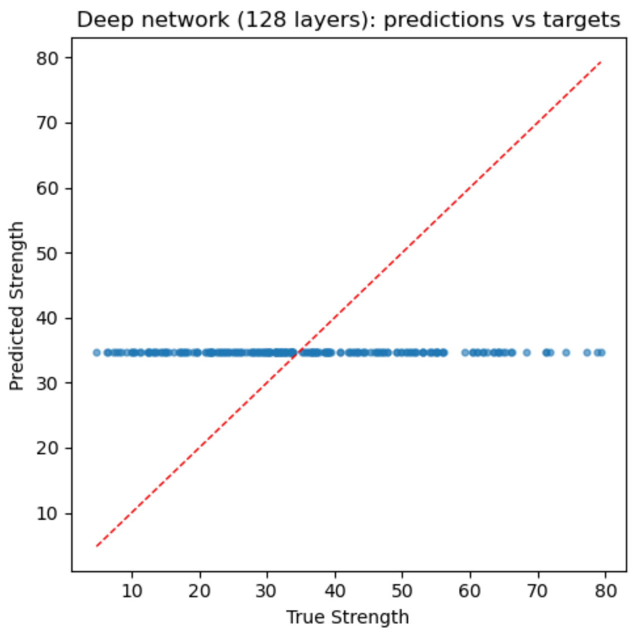


Figure 18: Predicted vs. true strength values for the 128-layer network. Validation loss: 283.0930824279785

The real problem of the vanishing gradients are clearly visible in Figure 18. The predictions of the 128-layer network stay to an almost constant value as shown in the scatter plot. This indicates that the model has failed to learn the underlying structure of the data which is not surprising since the gradients during the backpropagation are (almost) zero. Since almost all layers do not receive usable gradient information during training, the network is effectively unable to tuen its parameters and defaults to predicting a value close to the mean of the target distribution which is effectively almost the same as our baseline mean in earlier tasks. But this only gets it's impact when the depth of the network is really large, for example for our 16 layer network as it can be seen in Figure 19 the model still manages to learn the parameters in a meaningful way.

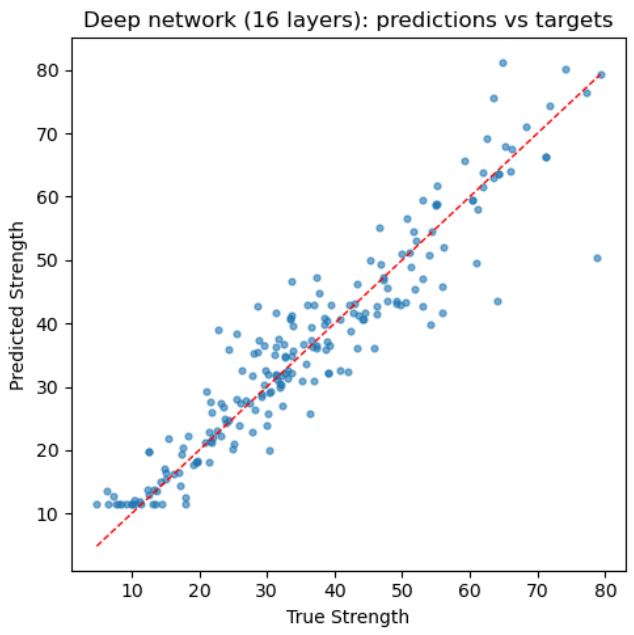


Figure 19: Predicted vs. true strength values for the network with 16 hidden layers. Validation loss: 31.778690338134766

Task 6

Based on the experiments in Task 2 and Task 3 we selected the configuration with hidden layers [128, 64], and adam optimizer as our final model. We combined the original training and validation sets into a single training set and retrained this model for 300 epochs with 64 batch size and learning rate of $\eta = 10^{-2}$.

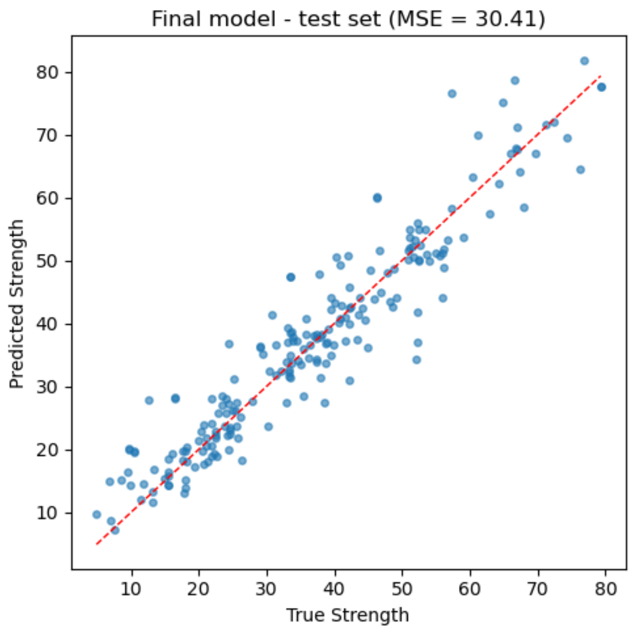


Figure 20: Final evaluation on train and test set with MSE = 30.4062

Evaluating the final model on the test set yields a mean squared error of 30.4062 as it can be seen in Figure 20, which is a lot lower than the baseline MSE obtained by predicting the mean strength = 288.60. The scatter plot of predicted versus true strengths on the test set shows that most points lie close to the diagonal (ground-truth target values = model predictions) which indicates accurate predictions across the full range of strengths. There is no strong bias where it consistently underestimates high or low strengths. So in overall we can conclude with the low test error and the scatter plot that our chosen neural network generalizes well to unseen data.