Neural Networks Homework 9

Sayeh Jarollahi (7073520, saja00006@stud.uni-saarland.de)
Mahsa Amani (7064006, maam00002@stud.uni-saarland.de)

January 29, 2025

Exercise 9.1

Proof. 1. Under the Gauss-Markov assumptions, an Ordinary Least Squares (OLS) estimator has the bias equals to 0, thus for a single model is unbiased and $\mathbb{E}[\hat{\beta}] = \beta$ (β represents the vector of true coefficients (parameters)).
In the Bagging Ensemble method, each model is trained on a subset of the data. OLS remains unbiased regardless of dataset size, yielding $\mathbb{E}[\hat{\beta}_i] = \beta$ for each subset. The ensemble average of these estimators is given by: $\hat{\beta}_{\text{ensemble}} = \frac{1}{m}\sum_{i=1}^{m}\hat{\beta}_i$. The expectation of the ensemble estimator is: $\mathbb{E}[\hat{\beta}_{\text{ensemble}}] = \frac{1}{m}\sum_{i=1}^{m}\mathbb{E}[\hat{\beta}_i] = \beta$, demonstrating that the ensemble is also unbiased.
In a Single Model, the variance of the OLS estimator for a dataset of size N is given by $\text{Var}(\hat{\beta}) \propto \frac{\sigma^2}{N}$.
For a Bagging Ensemble, each model is trained on $\frac{2}{3}N$ data points, resulting in an individual model variance of $\text{Var}(\hat{\beta}_i) \propto \frac{\sigma^2}{\frac{2}{3}N} = \frac{3\sigma^2}{2N}$. The variance of the ensemble average decreases by a factor of m, leading to $\text{Var}(\hat{\beta}_{\text{ensemble}}) = \frac{1}{m}\cdot\text{Var}(\hat{\beta}_i) \propto \frac{1}{m}\cdot\frac{3\sigma^2}{2N} = \frac{3\sigma^2}{2mN}$. Thus, the variance of the ensemble is lower than that of a single model by a factor of $\frac{1}{m}$.
To explain more simply, bagging leverages the averaging of predictions from multiple models trained on different subsets of data. While the bias remains constant due to OLS's inherent unbiasedness, the variance decreases as averaging multiple independent estimators reduces variability.

2. Bagging can effectively reduce variance, enhance robustness against overfitting, and improve predictive accuracy in high-variance scenarios or smaller datasets while managing outliers and noise. However, its effectiveness is hindered by high computational costs from training multiple networks and the diminished effective dataset size for each model, which can negatively impact performance in small datasets. Although bagging increases ensemble diversity, neural networks' resilience to minor data changes may limit its advantages over simpler models. Alternatives like Dropout often yield similar benefits at lower computational costs, making bagging more suitable for scenarios where performance improvements justify the additional expense.

3. Dropout and bagging aim to reduce variance by averaging multiple models. Dropout creates an ensemble effect within a single model by randomly deactivating neurons, while bagging trains multiple models explicitly. As dropout focuses on one model, it is more computationally efficient and primarily reduces variance by preventing overfitting. By introducing noise through random neuron deactivation, dropout regularizes the network, making it less sensitive to specific training data. It is particularly effective for high-variance models, such as deep networks or those trained on small datasets. However, dropout can slightly increase bias by limiting the network's capacity during training, as fewer neurons are active in each pass, potentially hindering the model's ability to fully fit the training data. Still, its regularization benefits generally outweigh this trade-off, especially when reducing variance is prioritized over minimizing bias. If a model is already underfitting (high bias), dropout may worsen performance by increasing bias further. It is computationally efficient and ideal for large, overparameterized neural networks prone to overfitting. In contrast, for smaller models or datasets, the advantages of dropout may diminish, and other regularization techniques like L1 and L2 may be sufficient without adding unnecessary complexity.

4. Co-adaptation occurs when multiple hidden units in a neural network develop dependencies on each other during training. Instead of each unit learning independent, robust, and meaningful features, the units work together in a highly specialized way to minimize the loss for the training data. This often results in overfitting, as the network memorizes specific patterns instead of generalizing, causing poor performance on new data. Dropout mitigates this by introducing randomness in the network during training. By making the presence of any given unit unreliable, dropout prevents hidden units from relying on specific other units to perform well. This encourages hidden units to develop more independent, generalizable features, ultimately enhancing model performance on unseen data.

5. In **standard dropout**, each neuron is randomly dropped (set to 0) during training with a probability of $1 - p$, where p is the retention probability. During inference, the full network is used without dropout, and the weights of each neuron are scaled by p to match the expected output from training. Without this scaling, the total activation during inference would exceed that of training, since all neurons are active.
In **inverted dropout**, neurons are still dropped with a probability of $1 - p$ during training, but the retained neurons are scaled by $\frac{1}{p}$. No scaling occurs during inference, allowing the network to perform a standard forward pass without dropout. This scaling during training ensures that expected activation levels remain consistent between training and inference, eliminating the need for additional scaling during inference and simplifying the process.
Both standard and inverted dropout are mathematically equivalent, as they yield the same expected activation for neurons during training and inference. The key difference is in the timing of the scaling: standard dropout applies it during inference, while inverted dropout applies it during training. Inverted dropout is often preferred for its simplicity during inference, and both methods effectively maintain consistent behavior between training and inference through appropriate scaling of neuron activations.

6. The dropout matrix elements R_{ij} are sampled from a Bernoulli distribution with param-

eter p, meaning $R_{ij} = 1$ with probability p and $R_{ij} = 0$ with probability $1 - p$. Using these properties, the expectation is simply p.

$$\mathbb{E}[R_{ij}] = 1 \times p + 0 \times (1 - p) = p$$

To prove $\mathbb{E}[R_{ij}R_{ik}] = \delta_{jk}p + (1 - \delta_{jk})p^2$, we have 2 cases:

1. If $j = k$, $\mathbb{E}[R_{ij}^2] = \mathbb{E}[R_{ij}] = p$.
2. If $j \neq k$, R_{ij} and R_{ik} are independent, so $\mathbb{E}[R_{ij}R_{ik}] = \mathbb{E}[R_{ij}] \cdot \mathbb{E}[R_{ik}] = p^2$.

Note that δ_{jk} is dependent on j and k.
The loss function with dropout is:

$$J(w; \{x_i\}_{i=1}^m, y) = \sum_{i=1}^{m} \left(y^{(i)} - \sum_{j=1}^{n} w_j R_{ij} x_j^{(i)} \right)^2$$

By taking the expectation over R_{ij} we have:

$$\mathbb{E}[J(w)] = \sum_{i=1}^{m} \mathbb{E}\left[\left(y^{(i)} - \sum_{j=1}^{n} w_j R_{ij} x_j^{(i)} \right)^2 \right]$$

$$\mathbb{E}[J(w)] = \sum_{i=1}^{m} \left[\mathbb{E}\left[\left(y^{(i)}\right)^2 \right] - 2y^{(i)} \mathbb{E}\left[\sum_{j=1}^{n} w_j R_{ij} x_j^{(i)} \right] + \mathbb{E}\left[\left(\sum_{j=1}^{n} w_j R_{ij} x_j^{(i)} \right)^2 \right] \right]$$

Using $\mathbb{E}[R_{ij}] = p$:

$$\mathbb{E}\left[\sum_{j=1}^{n} w_j R_{ij} x_j^{(i)} \right] = \sum_{j=1}^{n} p w_j x_j^{(i)}$$

So for the squared term:

$$\mathbb{E}\left[\left(\sum_{j=1}^{n} w_j R_{ij} x_j^{(i)} \right)^2 \right] = \sum_{j=1}^{n} w_j^2 x_j^{(i)2} \mathbb{E}[R_{ij}^2] + \sum_{j \neq k} w_j w_k x_j^{(i)} x_k^{(i)} \mathbb{E}[R_{ij}R_{ik}]$$

Using $\mathbb{E}[R_{ij}^2] = p$ and $\mathbb{E}[R_{ij}R_{ik}] = p^2$ for $j \neq k$, this becomes:

$$\mathbb{E}\left[\left(\sum_{j=1}^{n} w_j R_{ij} x_j^{(i)} \right)^2 \right] = p \sum_{j=1}^{n} w_j^2 x_j^{(i)2} + p^2 \sum_{j \neq k} w_j w_k x_j^{(i)} x_k^{(i)}$$

by combining these terms and substituting them into the expanded loss function, we have:

$$\mathbb{E}[J(w)] = \sum_{i=1}^{m} \left(y^{(i)} - p \sum_{j=1}^{n} w_j x_j^{(i)} \right)^2 + p(1 - p) \sum_{j=1}^{n} w_j^2 \sum_{i=1}^{m} x_j^{(i)2}$$

Rewriting this in matrix form:

$$\mathbb{E}[J(w)] = \|y - pXw\|^2 + p(1-p)w^\top Dw$$

where X is the $m \times n$ data matrix and D is a diagonal matrix with entries $D_{jj} = \sum_{i=1}^{m} x_j^{(i)2}$. To minimize this quadratic form, we take the its gradient with respect to w and set it to zero:

$$-2pX^\top(y - pXw) + 2p(1-p)Dw = 0$$

Simplify:

$$X^\top Xw + \frac{(1-p)}{p}Dw = X^\top y$$

Factor out w:

$$\left(X^\top X + \frac{(1-p)}{p}D\right)w = X^\top y$$

Finally the closed-form solution for the weight vector w is:

$$w = \left(X^\top X + \frac{(1-p)}{p}D\right)^{-1} X^\top y$$

The closed-form solution shows that dropout introduces a modified L_2 regularization term weighted by $\frac{(1-p)}{p}$, where features with larger magnitudes in D are penalized more strongly. □

Exercise 9.2

Proof. 1. GD computes the gradient of the loss function using the entire dataset at each step, making each iteration computationally expensive. While it converges smoothly, its slow speed makes it impractical for large datasets. In contrast, SGD calculates the gradient using a single randomly selected data point or a small batch, leading to greater computational efficiency. This stochastic approach can introduce noise, helping to avoid local minima and promoting better generalization, although it may cause more oscillation during training compared to GD.
2. Implicit regularization refers to the inherent biases of optimization algorithms, such as SGD, which direct the learning process toward solutions that generalize better, even without explicit regularization terms in the objective function. In SGD, this implicit regularizer penalizes the norm of the minibatch gradients and promotes solutions that balance loss landscape sharpness and gradient diversity. This approach improves test accuracy by favoring smoother, less sharp areas of the parameter space, especially when using moderate learning rates and small batch sizes.

3. The modified loss function in SGD is given by:

$$\tilde{C}_{\text{SGD}}(\omega) = C(\omega) + \frac{\epsilon}{4m} \sum_{k=0}^{m-1} \|\nabla C_k(\omega)\|^2$$

where $C(\omega)$ is the original loss, and the second term is the implicit regularizer. This regularizer penalizes the norm of minibatch gradients, and scales with the ratio of the learning rate ϵ to the batch size, effectively smoothing sharp regions in the loss landscape. With a small but finite learning rate, the implicit regularizer biases the optimization toward flatter minima, improving generalization and reducing the risk of overfitting.
4. Minibatch algorithms like SGD offer the advantage of parallelizability, allowing gradient computations for each minibatch to be distributed across multiple processors or GPUs. This simultaneous computation of minibatch gradients significantly reduces training time and optimizes the use of modern hardware. By dividing the dataset into smaller, independently processed chunks, minibatch algorithms achieve a balance between computational efficiency and convergence speed. □

Exercise 9.3

Proof. 1. a) The Taylor expansion for $f(\theta)$ around θ^k is:

$$f(\theta^{k+1}) \approx f(\theta^k) + \nabla_\theta f(\theta^k)^T(\theta^{k+1} - \theta^k) + \frac{1}{2}(\theta^{k+1} - \theta^k)^T H(\theta^{k+1} - \theta^k)$$

where H is the Hessian of $f(\theta)$ at θ^k.
Using the update rule $\theta^{k+1} = \theta^k - \alpha\nabla_\theta f(\theta^k)$, the change in the cost function becomes:

$$f(\theta^{k+1}) - f(\theta^k) \approx -\alpha\|\nabla_\theta f(\theta^k)\|^2 + \frac{\alpha^2}{2}\nabla_\theta f(\theta^k)^T H \nabla_\theta f(\theta^k)$$

For $f(\theta^{k+1})$ to increase:

$$-\alpha\|\nabla_\theta f(\theta^k)\|^2 + \frac{\alpha^2}{2}\nabla_\theta f(\theta^k)^T H \nabla_\theta f(\theta^k) > 0$$

In the second term, we use:

$$\lambda_{\text{eff}} = \frac{\nabla_\theta f(\theta^k)^T H \nabla_\theta f(\theta^k)}{\|\nabla_\theta f(\theta^k)\|^2}$$

and hence the equation will become:

$$-\alpha\|\nabla_\theta f(\theta^k)\|^2 + \frac{\alpha^2}{2}\|\nabla_\theta f(\theta^k)\|^2 \lambda_{\text{eff}} > 0$$

Simplify and rearrange:

$$\alpha\nabla_\theta f(\theta_k)^T H \nabla_\theta f(\theta_k) > 2\|\nabla_\theta f(\theta_k)\|^2$$

This condition is most likely to happen when α is too large or the Hessian H has large positive eigenvalues.
b)This can cause slow convergence and numerical problems. In this case the optimization path is heavily influenced by the eigenvalue corresponding to λ_{max}, leading to very small steps in directions related to the smaller eigenvalues.
c) In this case SGD with momentum can help. Momentum helps accelerate convergence in ill-conditioned problems by reducing oscillations along directions of large curvature.
2.
a) From the formula $\frac{\partial E_n}{\partial w_{11}^{(1)}} = \delta_1^{(2)} z_1^{(1)}$, we need $\delta_1^{(2)}$ and $z_1^{(1)}$.

$$z_1^{(1)} = x_1$$

$$\delta_1^{(2)} = h'(a_1^{(2)}) \sum_l w_{l1} \delta_l^{(3)}$$

We have to repeat backpropagate $\delta^{(3)}$ from $\delta^{(3)} = h'(\mathbf{a}^{(3)}) \odot (W^T \delta^{(4)})$
By repeating this process for all layers $k = N, N-1, \ldots, 2$, and substituting into the weight gradient formula:

$$\frac{\partial E_n}{\partial w_{11}^{(1)}} = x_1 \cdot h'(a_1^{(2)}) \cdot \sum_l w_{l1} \cdot \left(h'(a_l^{(3)}) \cdot \sum_m w_{ml} \cdot \delta_m^{(4)} \right)$$

Using matrix-vector notation, this becomes:

$$\frac{\partial E_n}{\partial w_{11}^{(1)}} = x_1 \cdot \left[\left(h'(\mathbf{a}^{(2)}) \odot (W^T (h'(\mathbf{a}^{(3)}) \odot (W^T \delta^{(4)}))) \right)_1 \right]$$

where the subscript 1 selects the first element of the resulting vector.
b) In backpropagation process, at the output layer we will have:

$$\delta^{(N+1)} = \nabla_{\mathbf{y}} E_n$$

For earlier layers, this delta will propagate:

$$\delta^{(k)} = W^T \delta^{(k+1)}$$

Since $\nabla_{\mathbf{y}} E_n$ is an eigenvector of W, we know:

$$W \nabla_{\mathbf{y}} E_n = \lambda \nabla_{\mathbf{y}} E_n$$

where λ is the corresponding eigenvalue. Therefore:

$$\delta^{(k)} = (W^T)^{N+1-k} \nabla_{\mathbf{y}} E_n$$

Since W^T and W share the same eigenvalues, this simplifies to:

$$\delta^{(k)} = \lambda^{N+1-k} \nabla_{\mathbf{y}} E_n$$

In other words by looking at the weight gradients:

$$\frac{\partial E_n}{\partial w_{11}^{(1)}} = \delta_1^{(2)} z_1^{(1)}$$

Substituting $\delta^{(2)}$:

$$\delta^{(2)} = W^T \delta^{(3)} = \lambda \delta^{(3)}$$

and further propagating:

$$\delta^{(3)} = W^T \delta^{(4)} = \lambda \delta^{(4)}$$

The result is:

$$\delta^{(k)} = \lambda^{N+1-k} \nabla_{\mathbf{y}} E_n$$

For $\frac{\partial E_n}{\partial w_{11}^{(1)}}$, we now have:

$$\frac{\partial E_n}{\partial w_{11}^{(1)}} = x_1 \cdot \lambda^{N-1} (\nabla_{\mathbf{y}} E_n)_1$$

where $(\nabla_{\mathbf{y}} E_n)_1$ is the first element of $\nabla_{\mathbf{y}} E_n$.
Now we have 2 cases:

1. $|\lambda| < 1$: λ^{N-1} decreases exponentially as N increases. This means the weight gradient $\frac{\partial E_n}{\partial w_{11}^{(1)}}$ becomes very small, especially for deep networks. This leads to vanishing gradient which makes the training of network hard.

2. $|\lambda| > 1$: λ^{N-1} grows exponentially with N. This means the weight gradient $\frac{\partial E_n}{\partial w_{11}^{(1)}}$ becomes very large, especially for deep networks. This leads to exploding gradients and the weights can diverge during training.

c) The weight gradient depends on the product of deltas across layers. For Sigmoid Activation ($\sigma(x)$), each layer introduces a factor of $h'(a_j^{(k)}) \leq 1/4$. Thus:

$$\delta_j^{(k)} \propto \left(\frac{1}{4}\right)^{N-k}$$

$\delta_j^{(k)}$ becomes exponentially small as N (the number of layers) increases, leading to vanishing gradients:

$$\frac{\partial E_n}{\partial w_{11}^{(1)}} \to 0 \text{ as } N \to \infty$$

And for Tanh Activation ($\tanh(x)$), each layer introduces a factor of $h'(a_j^{(k)}) \leq 1$, but typically $h'(a_j^{(k)})$ is much smaller than 1 for inputs far from 0 (e.g., due to saturation). While the vanishing gradient issue is less severe than sigmoid, it still exists.
d)

1. Exploding Gradients: Gradients can grow exponentially large during backpropagation in deep networks, leading to unstable training and divergence of the optimization process.

2. Overfitting: Neural networks with high capacity can memorize training data instead of generalizing to unseen data, resulting in poor performance on test datasets.

□