Stories from modern Machine Learning from the optimization perspective

Daniil Merkulov

Optimization for ML. Faculty of Computer Science. HSE University

Optimization for Neural Network training

Neural network is a function, that takes an input x and current set of weights (parameters) $\mathbf{w}$ and predicts some vector as an output. Note, that a variety of feed-forward neural networks could be represented as a series of linear transformations, followed by some nonlinear function (say, $\text{ReLU}(x)$ or sigmoid):

Optimization for Neural Network training

Neural network is a function, that takes an input x and current set of weights (parameters) $\mathbf{w}$ and predicts some vector as an output. Note, that a variety of feed-forward neural networks could be represented as a series of linear transformations, followed by some nonlinear function (say, ReLU (x) or sigmoid):

$$\mathcal{NN}(\mathbf{w}, x) = \sigma_L \circ w_L \circ \ldots \circ \sigma_1 \circ w_1 \circ x \qquad \mathbf{w} = (W_1, b_1, \ldots W_L, b_L),$$

Optimization for Neural Network training

Neural network is a function, that takes an input x and current set of weights (parameters) $\mathbf{w}$ and predicts some vector as an output. Note, that a variety of feed-forward neural networks could be represented as a series of linear transformations, followed by some nonlinear function (say, ReLU (x) or sigmoid):

$$\mathcal{NN}(\mathbf{w}, x) = \sigma_L \circ w_L \circ \ldots \circ \sigma_1 \circ w_1 \circ x \qquad \mathbf{w} = (W_1, b_1, \ldots W_L, b_L),$$

where L is the number of layers, σ_i - non-linear activation function, $w_i = W_i x + b_i$ - linear layer.

Optimization for Neural Network training

Neural network is a function, that takes an input x and current set of weights (parameters) $\mathbf{w}$ and predicts some vector as an output. Note, that a variety of feed-forward neural networks could be represented as a series of linear transformations, followed by some nonlinear function (say, ReLU (x) or sigmoid):

$$
\mathcal{NN}(\mathbf{w}, x) = \sigma_L \circ w_L \circ \ldots \circ \sigma_1 \circ w_1 \circ x \qquad \mathbf{w} = (W_1, b_1, \ldots W_L, b_L),
$$

where L is the number of layers, σ_i - non-linear activation function, $w_i = W_i x + b_i$ - linear layer.

Typically, we aim to find $\mathbf{w}$ in order to solve some problem (let say to be $\mathcal{NN}(\mathbf{w}, x_i) \sim y_i$ for some training data x_i, y_i). In order to do it, we solve the optimization problem:

Optimization for Neural Network training

Neural network is a function, that takes an input x and current set of weights (parameters) $\mathbf{w}$ and predicts some vector as an output. Note, that a variety of feed-forward neural networks could be represented as a series of linear transformations, followed by some nonlinear function (say, $\text{ReLU}(x)$ or sigmoid):

$$\mathcal{NN}(\mathbf{w}, x) = \sigma_L \circ w_L \circ \ldots \circ \sigma_1 \circ w_1 \circ x \qquad \mathbf{w} = (W_1, b_1, \ldots W_L, b_L),$$

where L is the number of layers, σ_i - non-linear activation function, $w_i = W_i x + b_i$ - linear layer.

Typically, we aim to find $\mathbf{w}$ in order to solve some problem (let say to be $\mathcal{NN}(\mathbf{w}, x_i) \sim y_i$ for some training data x_i, y_i). In order to do it, we solve the optimization problem:

$$L(\mathbf{w}, X, y) \to \min_{\mathbf{w}} \qquad \frac{1}{N} \sum_{i=1}^{N} l(\mathbf{w}, x_i, y_i) \to \min_{\mathbf{w}}$$

Loss functions

In the context of training neural networks, the loss function, denoted by $l(\mathbf{w}, x_i, y_i)$, measures the discrepancy between the predicted output $\mathcal{NN}(\mathbf{w}, x_i)$ and the true output y_i. The choice of the loss function can significantly influence the training process. Common loss functions include:

Mean Squared Error (MSE)

Used primarily for regression tasks. It computes the square of the difference between predicted and true values, averaged over all samples.

$$\text{MSE}(\mathbf{w}, X, y) = \frac{1}{N}\sum_{i=1}^{N}(\mathcal{NN}(\mathbf{w}, x_i) - y_i)^2$$

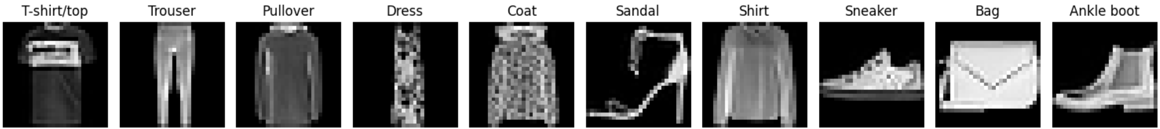
Cross-Entropy Loss

Typically used for classification tasks. It measures the dissimilarity between the true label distribution and the predictions, providing a probabilistic interpretation of classification.

$$\text{Cross-Entropy}(\mathbf{w}, X, y) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\log(\mathcal{NN}(\mathbf{w}, x_i)_c)$$

where $y_{i,c}$ is a binary indicator (0 or 1) if class label c is the correct classification for observation i, and C is the number of classes.

Simple example: Fashion MNIST classification problem



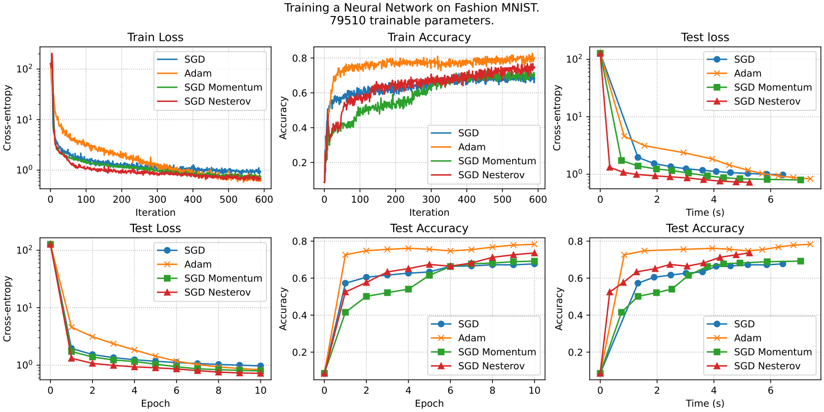


Figure 1: Open in colab

Visualizing loss surface of neural network via line projection

We denote the initial point as w_0, representing the weights of the neural network at initialization. The weights after training are denoted as $\hat{w}$.

Initially, we generate a random Gaussian direction $w_1 \in \mathbb{R}^p$, which inherits the magnitude of the original neural network weights for each parameter group. Subsequently, we sample the training and testing loss surfaces at points along the direction w_1, situated close to either w_0 or $\hat{w}$.

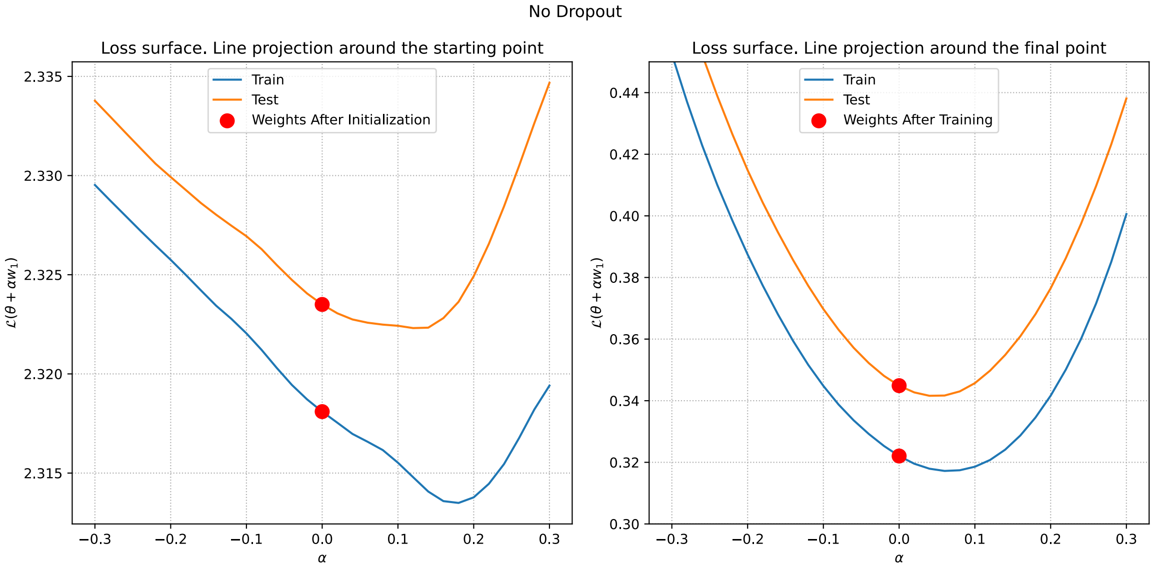
Mathematically, this involves evaluating:

$$L(\alpha) = L(w_0 + \alpha w_1), \text{ where } \alpha \in [-b, b].$$

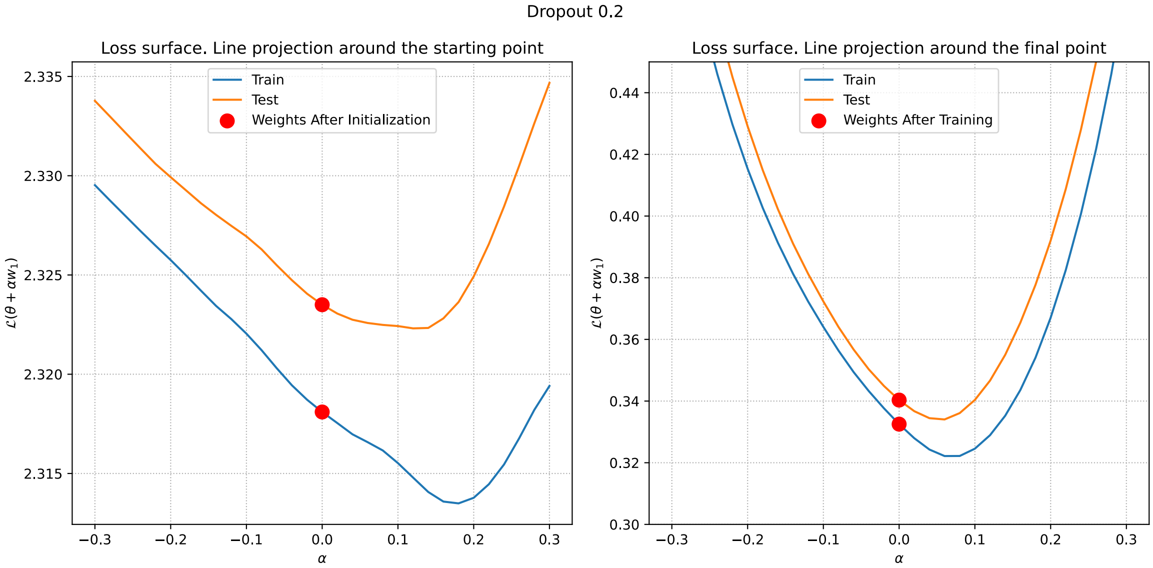
Here, α plays the role of a coordinate along the w_1 direction, and b stands for the bounds of interpolation. Visualizing $L(\alpha)$ enables us to project the p-dimensional surface onto a one-dimensional axis.

It is important to note that the characteristics of the resulting graph heavily rely on the chosen projection direction. It's not feasible to maintain the entirety of the information when transforming a space with 100,000 dimensions into a one-dimensional line through projection. However, certain properties can still be established. For instance, if $L(\alpha) \mid_{\alpha=0}$ is decreasing, this indicates that the point lies on a slope. Additionally, if the projection is non-convex, it implies that the original surface was not convex.

Visualizing loss surface of neural network



Visualizing loss surface of neural network



Plane projection

We can explore this idea further and draw the projection of the loss surface to the plane, which is defined by 2 random vectors. Note, that with 2 random gaussian vectors in the huge dimensional space are almost certainly orthogonal. So, as previously, we generate random normalized gaussian vectors $w_1, w_2 \in \mathbb{R}^p$ and evaluate the loss function

$$L(\alpha, \beta) = L(w_0 + \alpha w_1 + \beta w_2), \text{ where } \alpha, \beta \in [-b, b]^2.$$

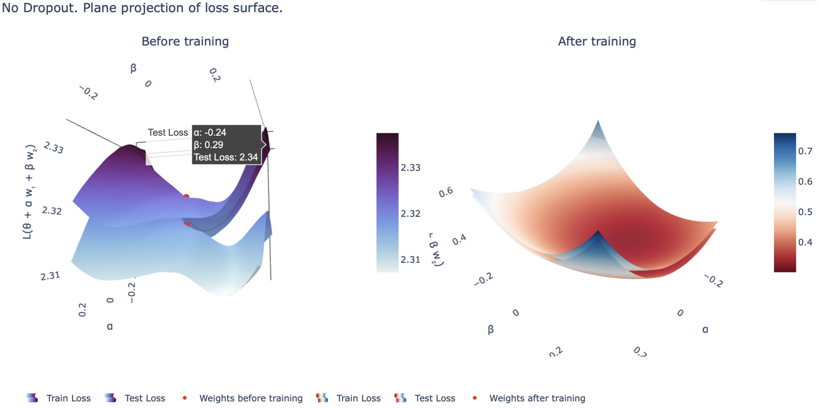


Figure 4: Open in colab

Can plane projections be useful? [1]

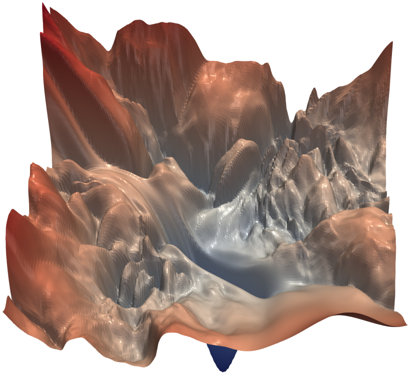


Figure 5: The loss surface of ResNet-56 without skip connections

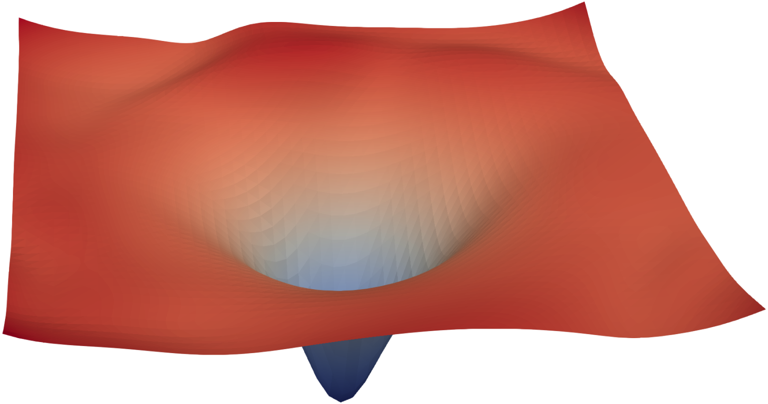


Figure 6: The loss surface of ResNet-56 with skip connections

[1] Visualizing the Loss Landscape of Neural Nets, Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, Tom Goldstein

Can plane projections be useful, really? [2]

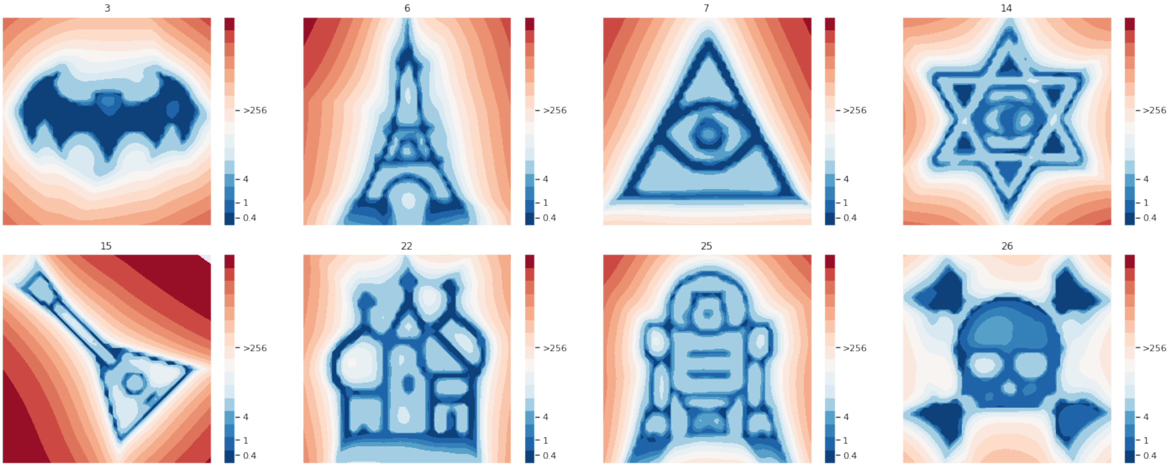


Figure 7: Examples of a loss landscape of a typical CNN model on FashionMNIST and CIFAR10 datasets found with MPO. Loss values are color-coded according to a logarithmic scale

[2] Loss Landscape Sightseeing with Multi-Point Optimization, Ivan Skorokhodov, Mikhail Burtsev

Impact of initialization [3]

Properly initializing a NN important. NN loss is highly nonconvex; optimizing it to attain a "good" solution hard, requires careful tuning.

- Don't initialize all weights to be the same — why?

Impact of initialization [3]

Properly initializing a NN important. NN loss is highly nonconvex; optimizing it to attain a "good" solution hard, requires careful tuning.

- Don't initialize all weights to be the same — why?
- Random: Initialize randomly, e.g., via the Gaussian $N(0, \sigma^2)$, where std σ depends on the number of neurons in a given layer. *Symmetry breaking*.

Impact of initialization [3]

Properly initializing a NN important. NN loss is highly nonconvex; optimizing it to attain a "good" solution hard, requires careful tuning.

- Don't initialize all weights to be the same — why?
- Random: Initialize randomly, e.g., via the Gaussian $N(0, \sigma^2)$, where std σ depends on the number of neurons in a given layer. *Symmetry breaking*.
- One can find more useful advices here

[3] On the importance of initialization and momentum in deep learning Ilya Sutskever, James Martens, George Dahl, Geoffrey Hinton

Impact of initialization [4]

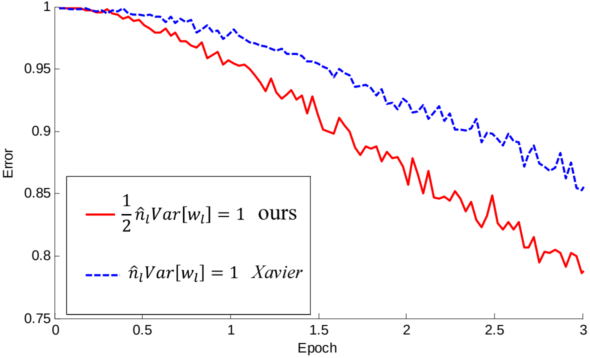


Figure 8: 22-layer ReLU net: good init converges faster

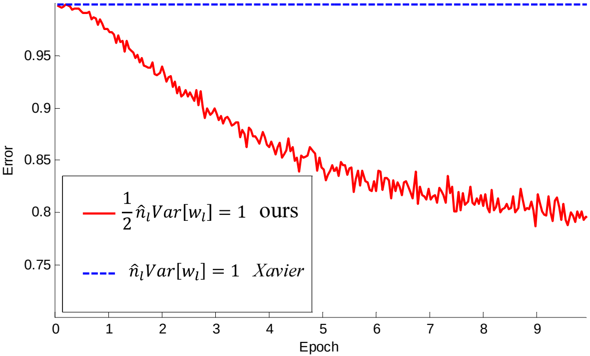


Figure 9: 30-layer ReLU net: good init is able to converge

[4] Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun

Grokking [5]

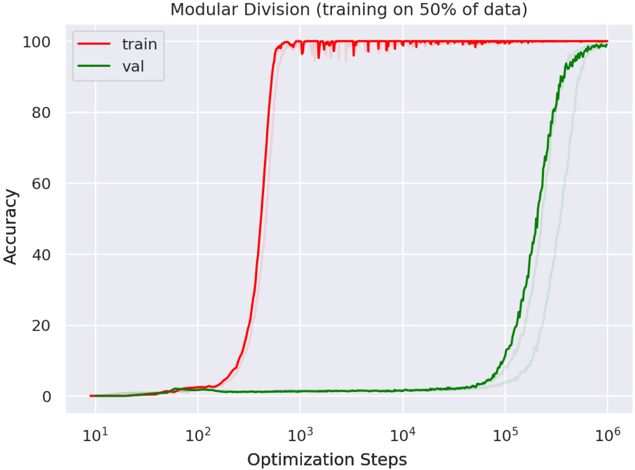
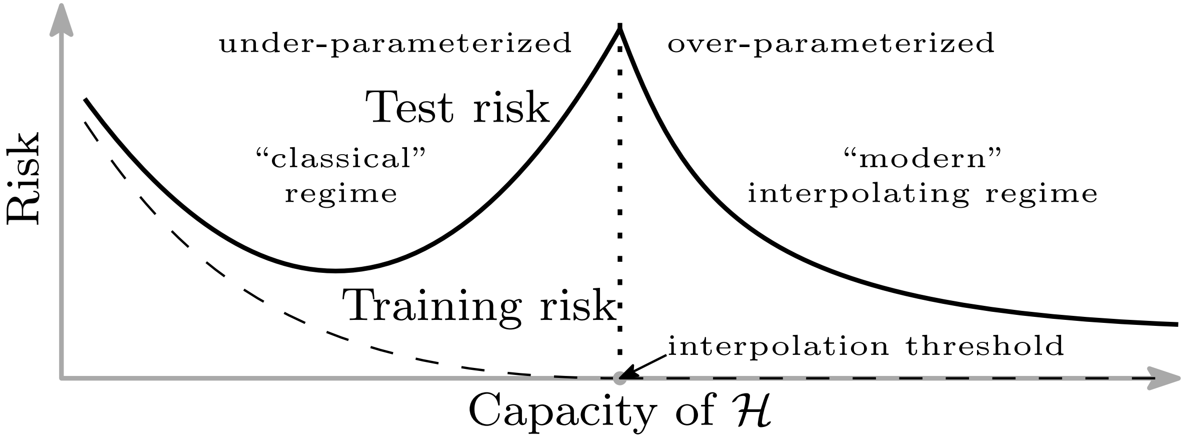


Figure 10: Training transformer with 2 layers, width 128, and 4 attention heads, with a total of about $4 \cdot 10^5$ non-embedding parameters. Reproduction of experiments ($\sim$ half an hour) is available here

[5] Grokking: Generalization Beyond Overfitting on Small Algorithmic Datasets, Alethea Power, Yuri Burda, Harri Edwards, Igor Babuschkin, Vedant Misra

Double Descent [6]



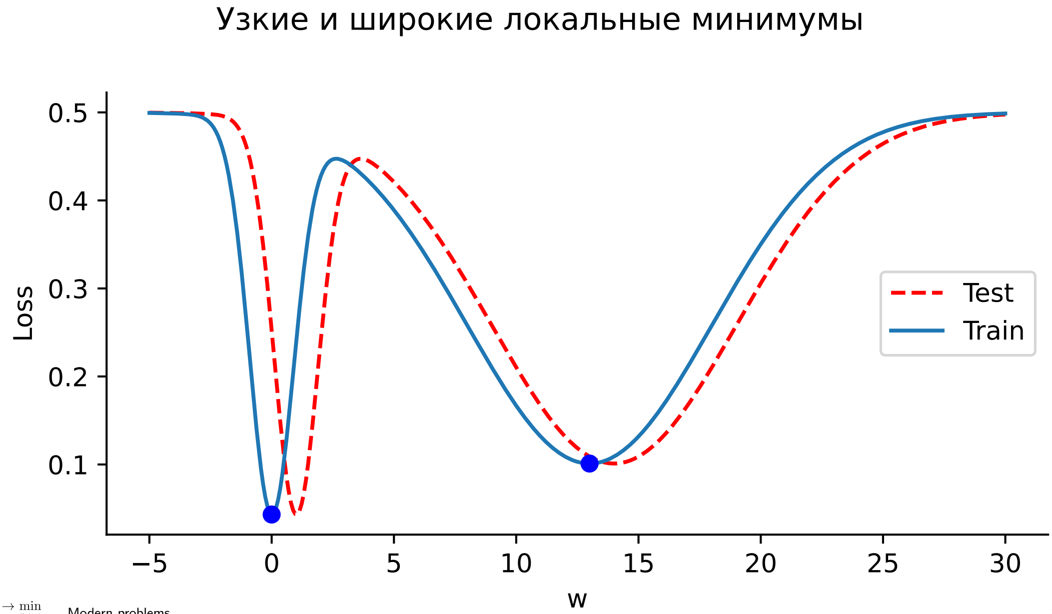
[6] Reconciling modern machine learning practice and the bias-variance trade-off, Mikhail Belkin, Daniel Hsu, Siyuan Ma, Soumik Mandal

Exponential learning rate

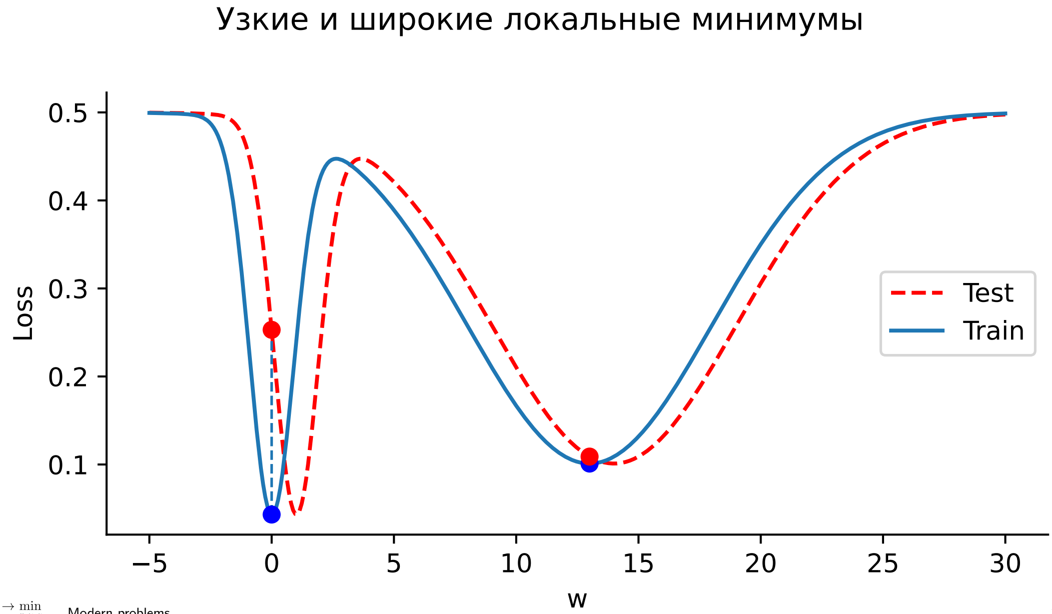
- Exponential Learning Rate Schedules for Deep Learning

Wide vs narrow local minima

Wide vs narrow local minima

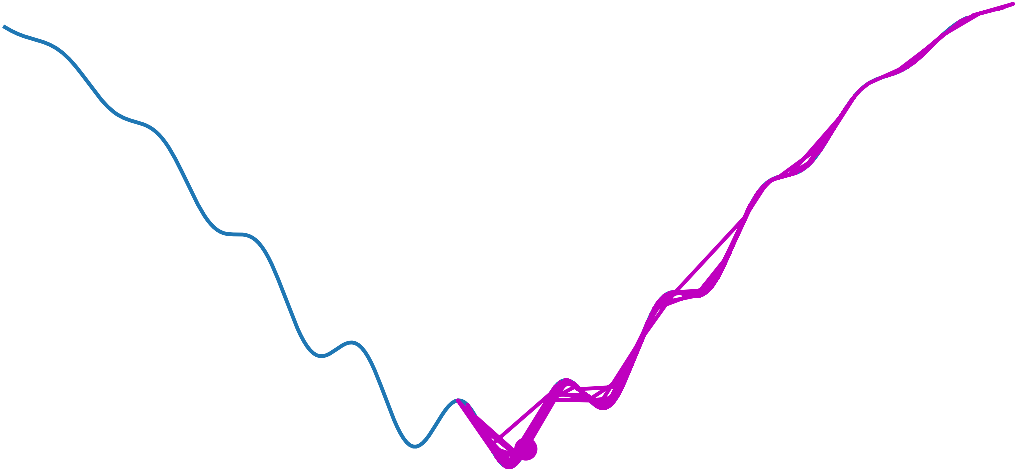


Wide vs narrow local minima



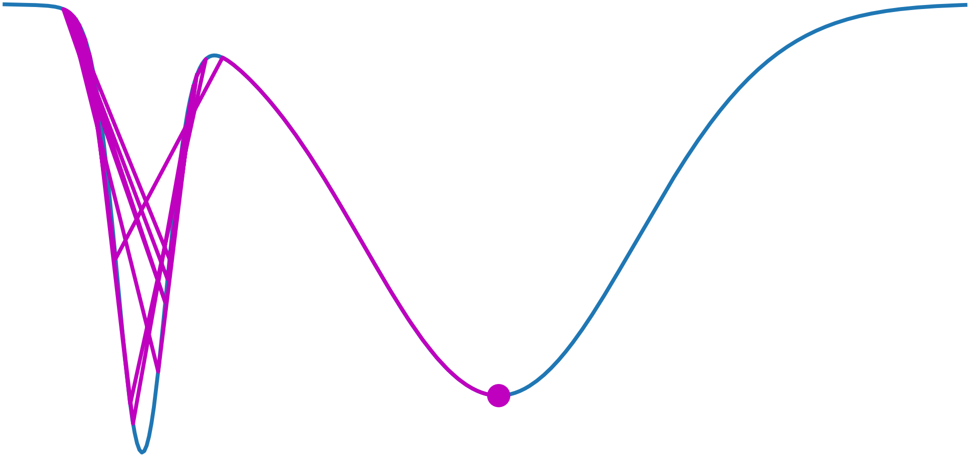
Stochasticity allows to escape local minima

Стохастический градиентный спуск
выпрыгивает из локальных минимумов



Local divergence can also be benefitial

Градиентный спуск с большим шагом
избегает узкого локального минимума



Gradient Vanishing/Exploding

- Multiplication of a chain of matrices in backprop

Gradient Vanishing/Exploding

- Multiplication of a chain of matrices in backprop
- If several of these matrices are “small” (i.e., norms < 1), when we multiply them, the gradient will decrease exponentially fast and tend to vanish (hurting learning in lower layers much more)

Gradient Vanishing/Exploding

- Multiplication of a chain of matrices in backprop
- If several of these matrices are “small” (i.e., norms < 1), when we multiply them, the gradient will decrease exponentially fast and tend to vanish (hurting learning in lower layers much more)
- Conversely, if several matrices have large norm, the gradient will tend to explode. In both cases, the gradients are unstable.

Gradient Vanishing/Exploding

- Multiplication of a chain of matrices in backprop
- If several of these matrices are "small" (i.e., norms < 1), when we multiply them, the gradient will decrease exponentially fast and tend to vanish (hurting learning in lower layers much more)
- Conversely, if several matrices have large norm, the gradient will tend to explode. In both cases, the gradients are unstable.
- Coping with unstable gradients poses several challenges, and must be dealt with to achieve good results.

Feedforward Architecture

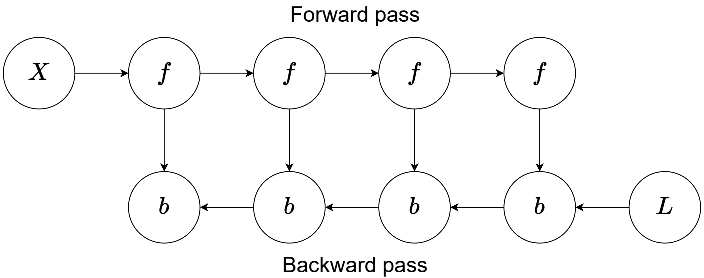


Figure 11: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The activations marked with an f. The gradient of the loss with respect to the activations and parameters marked with b.

Feedforward Architecture

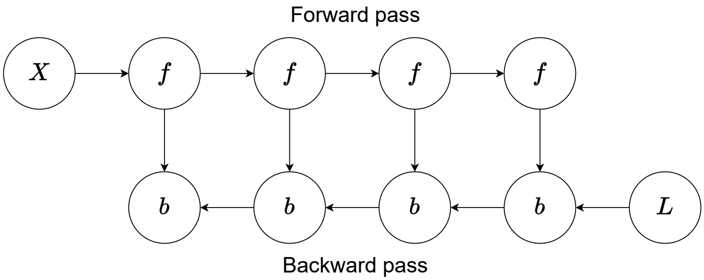


Figure 11: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The activations marked with an f. The gradient of the loss with respect to the activations and parameters marked with b.

! Important

The results obtained for the f nodes are needed to compute the b nodes.

Vanilla backpropagation

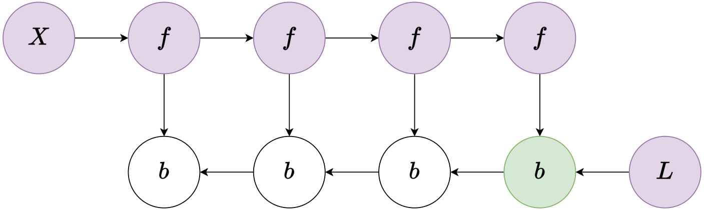


Figure 12: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

Vanilla backpropagation

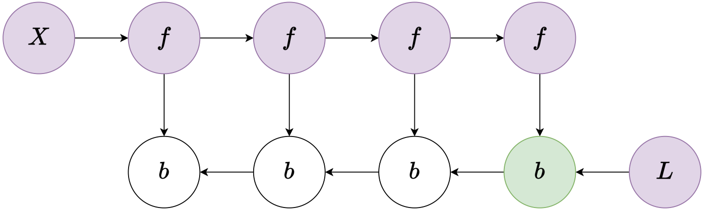


Figure 12: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

Vanilla backpropagation

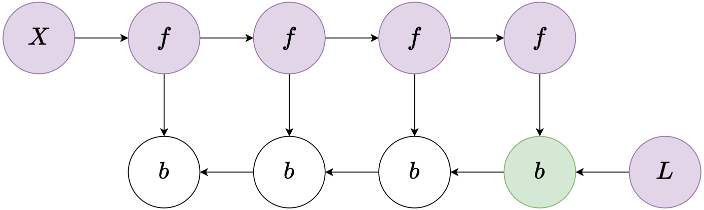


Figure 12: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

Vanilla backpropagation

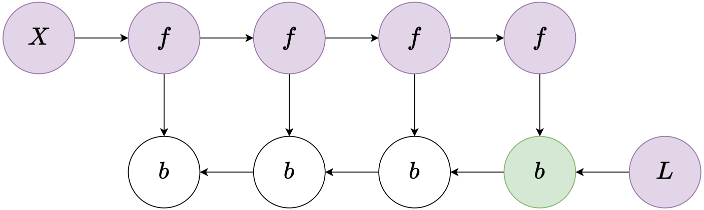


Figure 12: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

- Optimal in terms of computation: it only computes each node once.

Vanilla backpropagation

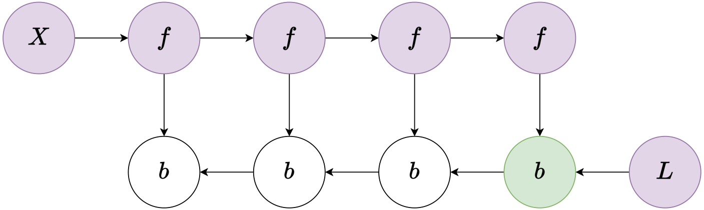


Figure 12: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

- Optimal in terms of computation: it only computes each node once.

Vanilla backpropagation

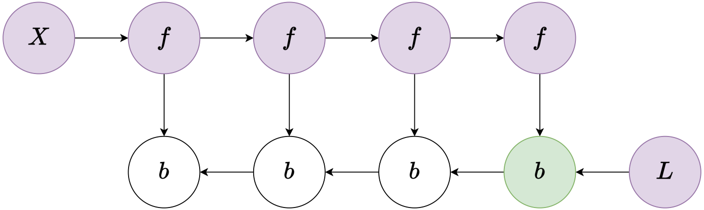


Figure 12: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

- Optimal in terms of computation: it only computes each node once.

- High memory usage. The memory usage grows linearly with the number of layers in the neural network.

Memory poor backpropagation

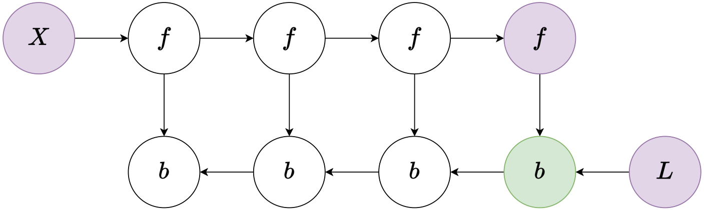


Figure 13: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

Memory poor backpropagation

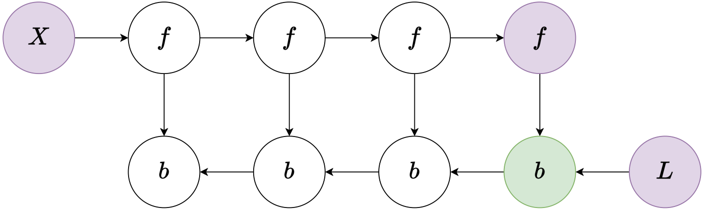


Figure 13: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

Memory poor backpropagation

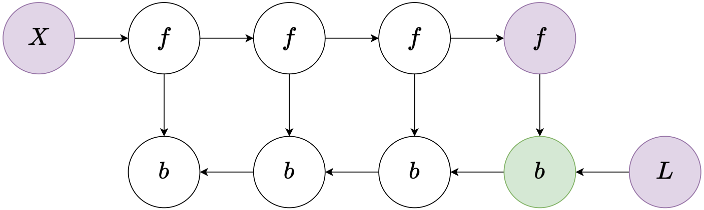


Figure 13: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

Memory poor backpropagation

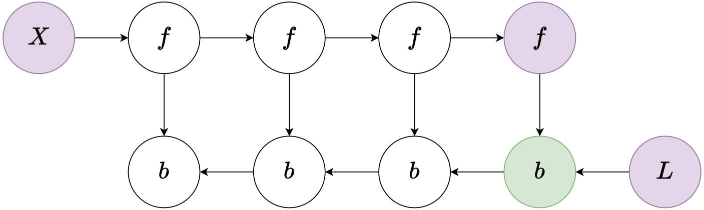


Figure 13: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

- Optimal in terms of memory: there is no need to store all activations in memory.

Memory poor backpropagation

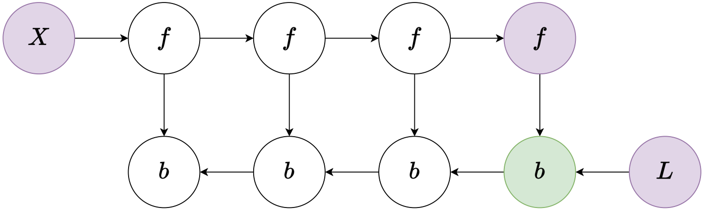


Figure 13: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

- Optimal in terms of memory: there is no need to store all activations in memory.

Memory poor backpropagation

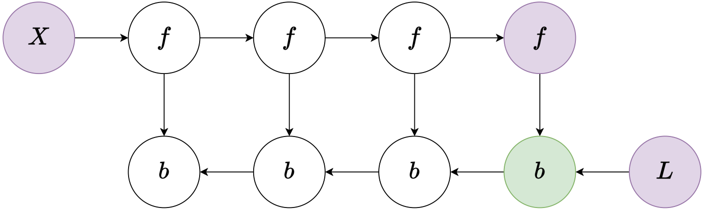


Figure 13: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

- Optimal in terms of memory: there is no need to store all activations in memory.

- Computationally inefficient. The number of node evaluations scales with n^2, whereas it vanilla backprop scaled as n: each of the n nodes is recomputed on the order of n times.

Checkpointed backpropagation

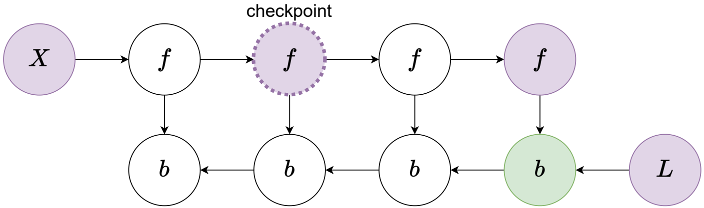


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

Checkpointed backpropagation

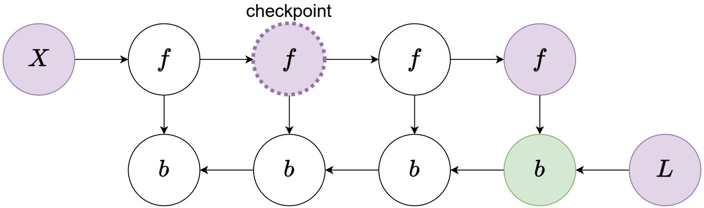


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

Checkpointed backpropagation

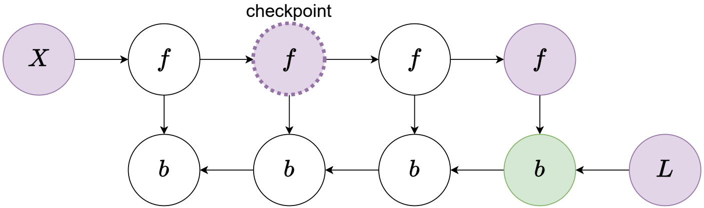


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

Checkpointed backpropagation

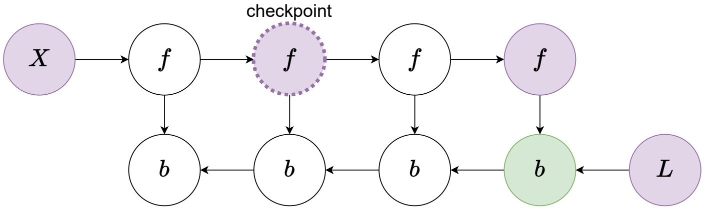


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

- Faster recalculation of activations f. We only need to recompute the nodes between a b node and the last checkpoint preceding it when computing that b node during backprop.

Checkpointed backpropagation

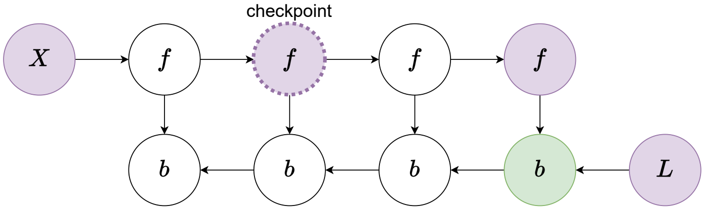


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

- Faster recalculation of activations f. We only need to recompute the nodes between a b node and the last checkpoint preceding it when computing that b node during backprop.

Checkpointed backpropagation

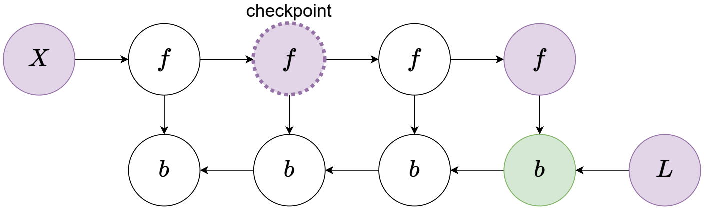


Figure 14: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

- Faster recalculation of activations f. We only need to recompute the nodes between a b node and the last checkpoint preceding it when computing that b node during backprop.

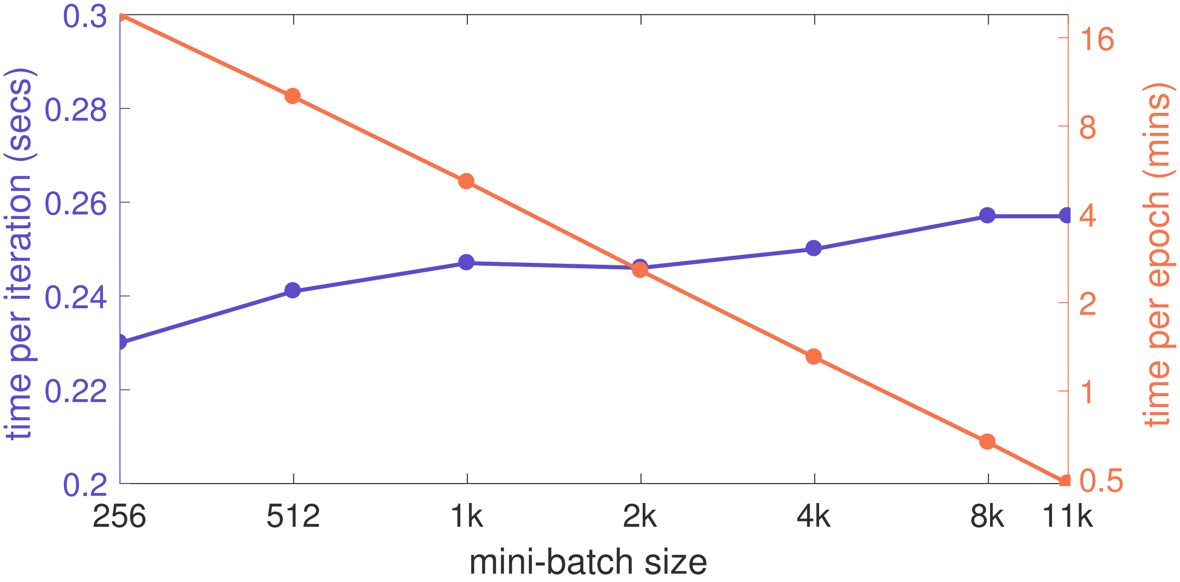
- Memory consumption depends on the number of checkpoints. More effective then **vanilla** approach.

Gradient checkpointing visualization

The animated visualization of the above approaches

An example of using a gradient checkpointing

Large batch training



Large batch training