is shown, [LIP-III, Thm. 2.1]. Here, $\mathcal{X}_s$ denote modified Poisson point processes, over which the expectation is taken. The distance $d_{\varepsilon_s,\mathcal{X}_s}(0, se_1)$ is basically a graph distance with unit weights, connecting 0 and the point se_1. We observe that this term is reminiscent of the dominating term Eq. (3.36) we have in [LIP-III]. In fact, employing the same strategy as displayed in the previous paragraphs, in [LIP-III] we are able to show a rate in the case $\varepsilon_n \sim \delta_n$.

Chapter 4

Robust and Sparse Supervised Learning

In this chapter we present the topics discussed in the works [CLIP; FNO; BREG-I] which are reprinted in **??**. Compared to the previous chapter we are now in the setting of supervised learning as described in Section 2.2. As already mentioned before, we focus on neural networks $f_\theta : \mathcal{X} \to \mathcal{Y}$ parameterized by $\theta \in \Theta$, where Θ is some parameter space. Here, our two main question arise, namely:

- **Input robustness**: how robust is $x \mapsto f_\theta(x)$ w.r.t. input perturbations?
- **Parameter sparsity**: how can we obtain sparse parameters $\theta \in \Theta$?

Section 4.1: Setting

Section 4.2: [CLIP] | Section 4.3: [FNO] | Section 4.4: [BREG-I]

Therefore, conceptually we again highlight the keyword **sparsity** and additionally **robustness**. In [CLIP] we consider robustness under adversarial perturbations. Here, the input is *attacked* on purpose to confuse a neural network classifier and therefore worsen its performance. In order to obtain a classifier that is less vulnerable against such attacks we propose an optimization strategy that selects parameters yielding a more robust network. In Section 4.2 we comment on the topic and the contribution in more detail.

A different kind of input robustness is considered in [FNO]. In the setting of image classification, images are modeled as functions on a continuum domain that need to be discretized in order to represent them on a machine. However, this discretization is usually arbitrary and not inherent to the object of interest. Therefore, it is natural to assume that the output of the network should be independent of this discretization, also referred to as resolution, hence we consider robustness w.r.t. resolution changes. We remark on this and the publication in Section 4.3.

Concerning computational performance and memory storage of the neural network we focus on sparsity of the parameters $\theta \in \Theta$. In [BREG-I] we propose a sparse optimization strategy based on Bregman iterations, which employs L^1 type penalties to promote sparsity. The conceptual difference between the existing algorithm and our proposal is the stochastic component in the gradient. Our theoretical results prove decay in loss and convergence of the iterates. Finally, we also conduct numerical experiments that demonstrate the efficiency of the method. We refer to Section 4.4 for a detailed explanation.

Before, we start with the exposition on the mentioned works, we briefly review the common supervised learning framework. Building upon the basic observations in Chapter 4 we give a slightly more detailed introduction in Section 4.1.

4.1. Setting

We are given a finite training set $\mathcal{T} \subset \mathcal{X} \times \mathcal{Y}$. For a family of functions $f_\theta : \mathcal{X} \to \mathcal{Y}$ parameterized by $\theta \in \Theta$, we consider the empirical minimization

$$\min_{\theta \in \theta} \mathcal{L}(\theta),$$

where for a function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ we define

$$\mathcal{L}(\theta) := \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \ell(f_\theta(x), y). \tag{4.1}$$

Remark 4.1. Assuming that $\mathcal{T}$ is sampled from a joint distribution $P_{\mathcal{X},\mathcal{Y}}$ on $\mathcal{X} \times \mathcal{Y}$ this approximates the computational infeasible population risk minimization problem

$$\inf_{\theta\in\Theta} \int_{\mathcal{X}\times\mathcal{Y}} \ell(f_\theta(x), y) d\pi(x, y).$$

△

In the following we provide two important choices for the function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$.

Example 4.2 (MSE). For image denoising problems, we often choose $\mathcal{X} = \mathcal{Y} = [0, 1]^{N\times M}$, assuming only one color channel for simplicity. In this context the mean squared L^2 error (MSE), defined as

$$\ell(\bar{y}, y) := \frac{1}{N \cdot M} \|\bar{y} - y\|^2$$

is commonly employed. This loss function could however also be employed for classification problems.

Example 4.3 (Cross-Entropy). For classification problems one often chooses the *cross-entropy* or *negative log-likelihood* loss, [Goo52]. For two discrete probability

distributions, $p, q : \{1, \dots, C\} \to \mathbb{R}$ one defines

$$H(p, q) := -\sum_{c=1}^{C} p_c \cdot \log(q_c)$$

see, e.g., [COR98]. Assuming that $\mathcal{Y} = \Delta^C$ this allows to choose $\ell(\bar{y}, y) := H(y, \bar{y})$. If the network only maps to $\mathbb{R}^C$ one often additionally inserts a soft-max function $Q(y)_c := \exp(y_c)/\sum_c \exp(y_c)$ (see [Bol68]) and then sets

$$\ell(\bar{y}, y) := H(y, Q(\bar{y})).$$

In the case, where the output is given as labels $y \in \{1, \dots, C\}$ one defines

$$\ell(\bar{y}, y) := H(y_{\text{oh}}, Q(\bar{y})) = -\log(Q(\bar{y}))$$

employing the one-hot notation from Chapter 2.

4.1.1. Network Architectures

In this thesis we focus on feed-forward neural networks, i.e., we consider layers of the form

$$\Phi(w, W, b)(z) := wz + \sigma(Wz + b) \tag{4.2}$$

where $w \in \mathbb{R}$ models a residual connection, $W \in \mathbb{R}^{n \times n}$ is a weight matrix, $b \in \mathbb{R}^n$ a bias vector and $z \in \mathbb{R}^n$ is the input. We consider a concatenation of $L \in \mathbb{N}$ of such layers, which then forms a neural network

$$f_\theta = \Phi_L \circ \dots \circ \Phi_1$$

with parameters $\theta = ((W_1, b_1, w_1) \dots, (W_L, b_L, w_L)) \in \Theta$ and layers $\Phi_i := \Phi(w_i, W_i, b_i)$. If there is no residual part, i.e., $w = 0$, then we also allow dimensional changes in each layer, i.e., $z \in \mathbb{R}^n, W \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m$.

MLP In the easiest case we consider a perceptron [Ros58], which models a fully connected layer, i.e. every entry W_{ij} of the weight matrix is a parameter that is optimized in the training process.

Convolutions Especially important for visual tasks are convolutional layers. Here, we take a kernel $k \in \mathbb{R}^{M \times M}$ and define the application of $W = W(k)$ as

$$Wz = k * z$$

with $*$ denoting the convolution. We refer to Section 4.3 for the concrete definition of this operation. Typically, the input is of the form $z \in \mathbb{R}^{K_{\text{in}} \times N \times M}$, where K_{in} denotes the

number of input channels. The layer is then a mapping $\Phi : \mathbb{R}^{K_{\text{in}} \times N \times M} \to \mathbb{R}^{K_{\text{out}} \times N \times M}$, where K_{out} denotes the number of output channels, which can be realized by different kernels k_{ij}. The application of W to an input z is defined as

$$(Wz)_{j,:,:} := \sum_{i=1}^{K_{\text{in}}} k_{ij} * z.$$

ResNets Often we also consider a residual component as displayed in Eq. (4.2) with the term $w\, z$. The idea of adding this residual component was first introduced in [SGS15] with a learnable parameter $w \in \mathbb{R}$ and later popularized in [He+16a] by fixing $w = 1$, see also [He+16b], which then yields the celebrated ResNet architecture. In the following applications we both consider the case where $w = 1$ is fixed, but also the possibility of learning the parameter $w \in \mathbb{R}$ in [BREG-II].

4.1.2. Gradient Computation and Stochastic Gradient Descent

Training a neural network requires solving an optimization problem w.r.t. to the parameters $\theta \in \Theta$. In this work we only focus on first order methods, however both zero [Rie22; Pin+17; Car+21] and second order methods [Mar10] have been successfully applied in this context. Employing first order methods, requires evaluating the gradient $\nabla_\theta \mathcal{L}$, however in this scenario it is not common to compute the full gradient but rather to have a gradient estimator. This estimator is usually obtained by randomly dividing the train set $\mathcal{T}$ into disjoint mini-batches $B_1 \cup \ldots \cup B_b = \mathcal{T}$ and then successively computing the gradient of the mini-batch loss

$$\mathcal{L}(\theta; B) := \frac{1}{|B_i|} \sum_{(x,y) \in B_i} \ell(f_\theta(x), y).$$

Iterating over all batches $i = 1, \ldots, b$ is referred to as one epoch. From a mathematical point of view, this yields stochastic optimization methods, since in each step the true gradient is replaced by an estimator. In the abstract setting we let $(\Omega, F, \mathbb{P})$ be a probability space and consider a function $g : \Theta \times \Omega \to \Theta$ as an unbiased estimator of $\nabla \mathcal{L}$, i.e.

$$\mathbb{E}\left[g(\theta; \omega)\right] = \nabla \mathcal{L}(\theta) \text{ for all } \theta \in \Theta.$$

Most notably this method transforms the standard gradient descent update [Cau+47]

$$\theta^{(k+1)} = \theta^{(k)} - \tau^{(k)} \nabla \mathcal{L}(\theta^{(k)}),$$

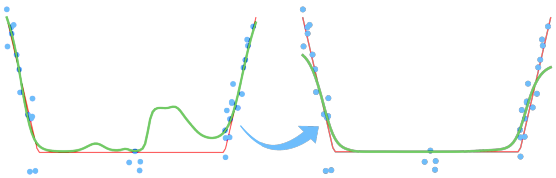
to *stochastic* gradient descent [RM51]

$$\begin{aligned} &\text{draw } \omega^{(k)} \text{ from } \Omega \text{ using the law of } \mathbb{P}, \\ &g^{(k)} := g(\theta^{(k)}; \omega^{(k)}), \\ &\theta^{(k+1)} := \theta^{(k)} - \tau^{(k)} g^{(k)}. \end{aligned}$$

4.2. Adversarial Stability via Lipschitz Training: [CLIP]

In the following we consider classification problems with $C \in \mathbb{N}$ different classes and functions $f : \mathcal{X} \to \Delta^C$ for which we denote by

$$f^{\mathrm{MAP}}(x) := \operatorname{argmax}_c f(x)_c$$



the maximum a posteriori estimation for an input $x \in \mathcal{X}$. The capabilities of neural networks to perform classification tasks on unseen data—i.e. inputs $x \in \mathcal{X}$ that are not part of the training data—are impressive and the reason for their popularity. However, it has been noticed in [GSS14] that it is relatively easy to "fool" classification networks in the following sense:

Given a classification network and a human classifier $f_\theta, h : \mathcal{X} \to \Delta^C$, and an input $x \in \mathcal{X}$ such that $f_\theta^{\mathrm{MAP}}(x) = h^{\mathrm{MAP}}(x)$. An adversarial example is an input $\bar{x} \in \mathcal{X}$ that is close to x, but we have that

$$f_\theta^{\mathrm{MAP}}(\bar{x}) \neq h^{\mathrm{MAP}}(\bar{x}) = h^{\mathrm{MAP}}(x).$$

The vague concept of a *human classifier*, incorporates the intuition of the classification problem as a human would solve it. Assuming that we are given data $\mathcal{T} \subset \mathcal{X} \times \mathcal{Y}$ that is sampled i.i.d. from a joint distribution $P_{\mathcal{X},\mathcal{Y}}$ this function could also be chosen as $h(x)_c := P_{\mathcal{X},\mathcal{Y}}(c|x)$. The term "close" describes that the difference between the two images $x, \bar{x}$ should be visually imperceptible.

Remark 4.4. Some authors argue that such instabilities are inherent to classification problems solved via a data-driven approach [Sha+18; FFF18]. From this point of view, trying to defend against these instabilities is not necessarily desirable. However, we do not consider this interpretation in the following. △

What are Adversarial Examples Formally? In order to formalize this idea, we omit any influence of the typically unavailable function h and instead consider $\bar{x}$ adversarial if $f_\theta^{\mathrm{MAP}}(x) \neq f_\theta^{\mathrm{MAP}}(\bar{x})$. This interpretation only makes sense, if we assume that the output of $f_\theta^{\mathrm{MAP}}(x)$ is *correct*, which can only be easily checked on labeled data [Bun+23]. Furthermore, in this work we assume that $\mathcal{X}$ models an image space $\mathcal{X} = \mathbb{R}^{K \times N \times M}$, with $K, N, M \in \mathbb{N}$ and choose a distance $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_0^+$ to measure the distance between points in $\mathcal{X}$.

Remark 4.5. In most of our examples, we choose $d(\cdot, \cdot) = \|\cdot - \cdot\|_p$. Employing a L^p norm in an image context—via pixel-wise comparisons—can be an unfavorable choice. Images that appear very different visually, can have a smaller L^p distance, than images that visually appear similar, see e.g. [Sta+21, Fig. 16]. However, it is easy to evaluate, which is crucial for most applications. In the absence of a better criterion, being close

to the original input in the L^p distance, is a practical way to decide if $\bar{x}$ is an admissible adversarial example. △

Employing these simplifications we then say that $\bar{x} \in \mathcal{X}$ is adversarial, if

$$d(x, \bar{x}) \leq \varepsilon \qquad \text{and} \qquad f_\theta^{\text{MAP}}(x) \neq f_\theta^{\text{MAP}}(\bar{x}),$$

where ε is called the *adversarial budget*. This parameter controls how far $\tilde{x}$ can be from the original point x to be still considered adversarial. Here, we also have to decide how much we trust the metric $d(\cdot, \cdot)$ as a measure of closeness.

Types of Adversarial Examples Typically, adversarial examples are created from the clean image x via some distortion $\delta \in \mathbb{R}^s$. Together with an application map $T : \mathcal{X} \times \mathbb{R}^s \to \mathcal{X}$ one then obtains $\bar{x} = T(x, \delta)$ as the adversarial example. In this formulation one can alternatively employ the criterion $\|\delta\| \leq \varepsilon$ to decide, whether $T(x, \delta)$ is an adversarial example. We only list some of the approaches below:

- **Addition**: The most well-known examples are created by $T(x, \delta) := x + \delta$, i.e. $\bar{x} = x + \delta$. Here, it is important to note that typically images are assumed to have values between 0 and 1, i.e. $\mathcal{X} = [0, 1]^{K \times N \times M}$, therefore it is important to also ensure that $\bar{x} \in \mathcal{X}$.

- **Translation and Rotation**: Simple geometric transformations—that are unnoticeable for a human classifier—are quite effective to "fool" neural networks. Employing translations $T_{\text{t}} : \mathcal{X} \times \mathcal{X} \to \mathcal{X}$ one has to choose the behavior on the boundary such that one obtains a valid image. The same holds true for rotations $T_{\text{r}} : \mathcal{X} \times [-\pi, \pi] \to \mathcal{X}$. In this case $\delta \in [-\pi, \pi]$ models the angle of the rotation and yields an admissible adversarial example, if $|\delta| \leq \varepsilon$. Here, we see that this formulation looses some expressivity. Consider the MNIST dataset [LC10], where the task is to classify handwritten digits from 0 to 9:
  - We see that only $\varepsilon < \pi/2$ makes sense, otherwise the number "6" could be always transformed to the number "9" and vice versa.
  - However, considering the number "0" rotations above the angle $\pi/2$ can definitely yield proper adversarial examples $\bar{x}$.

  We refer to [Eng+18] for a study on these types of adversarial examples.

- **Change of Basis**: As explored in [Guo+17] one can consider a different orthonormal basis of the image space $\mathbb{R}^{K \times N \times M}$ and then perform the attack w.r.t. this basis. The map $T : \mathcal{X} \times \mathbb{R}^{K \times N \times M} \to \mathcal{X}$ first obtains the different representation of x, then adds the coefficients of δ and then maps back to the original basis. This is only meaningful, if one restricts certain coefficients of δ to be zero in the alternative basis. For example, the discrete cosine transform ([ANR74]) has been applied successfully in this context [Guo+17].

In the following we restrict ourselves to the additive case and only consider examples $\bar{x} = x + \delta$.

Finding Adversarial Examples Employing a loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ the task of finding an adversarial examples $\bar{x} = x + \delta \in \mathcal{X}$ can be relaxed via solving the problem

$$\max_{\bar{x} \in \mathcal{X} : d(x,\bar{x}) \leq \varepsilon} \ell(y, f_\theta(\bar{x})) = \max_{\delta : \|\delta\|_d \leq \delta, x+\delta \in \mathcal{X}} \ell(y, f_\theta(x+\delta)), \tag{4.3}$$

where $y = f_\theta(x)$. Solving this problem is referred to as *attacking* the network f. More precisely, this is a so-called *untargeted* attack, since we do not prescribe $\bar{x}$ to realize any special output as long as it confuses the network. Opposed to this, there are so called *targeted* attacks. Here, we pick $c^{\text{adv}} \neq f_\theta^{\text{MAP}}(x)$ and then want to find $\bar{x}$ such that $f_\theta^{\text{MAP}}(\bar{x}) = c^{\text{adv}}$, which yields the problem

$$\min_{\bar{x} \in \mathcal{X} : d(x,\bar{x}) \leq \varepsilon} \ell(c_{\text{oh}}^{\text{adv}}, f_\theta(\bar{x})).$$

Conceptually, this problem is similar to Eq. (4.3) and differs only by a change of sign and a different reference label. A popular method to solve Eq. (4.3) is the projected sign gradient ascent iteration [KGB+16],

$$\bar{x}^{(k+1)} = \text{Proj}_d(\bar{x}^{(k)} + \tau \, \text{sign}(\nabla_{\bar{x}} \ell(f_\theta(x), f_\theta(\bar{x}^{(k)})))).$$

Here, Proj_d denotes the projection onto the set $\mathcal{X} \cap B_{d,\varepsilon}(x)$, where $B_{d,\varepsilon}(x)$ is the ball with radius ε around x, w.r.t. to the distance d. Performing only one step of this iteration yields the fast gradient sign method [GSS14]. There is a wide variety of these so-called gradient-based open-box attacks [Yua+19], i.e., methods that assume that the gradient of the model is available. In more realistic scenarios, this might not be the case. Attacks that do not employ the actual gradient of the model to attack are called *closed box* attacks [Ily+18], which are not part of this thesis.

Defending against Adversarial Attacks We consider the question of finding parameters $\theta \in \Theta$ such that corresponding model f_θ is *adversarially robust*, i.e., is less vulnerable to attacks. Therefore, we want the attack problem in Eq. (4.3) to be hard to solve. This intuition leads to the optimization problem

$$\min_{\theta \in \Theta} \sum_{(x,y) \in \mathcal{T}} \max_{\bar{x} : d(x,\bar{x}) \leq \varepsilon} \ell(f_\theta(\bar{x}), y)$$

which is known as the *adversarial training* formulation [KGB16; Mad+17]. From a distributionally robust optimization point of view, this problem relates to

$$\min_{\theta \in \Theta} \max_{\tilde{P} : D(P_{\mathcal{X},\mathcal{Y}}, \tilde{P}) \leq \varepsilon} \int_{\mathcal{X} \times \mathcal{Y}} \ell(f_\theta(x), y) \, d\tilde{P}(x, y)$$

where D denotes some distance on the space of probability distributions, see [BGM23]. Employing a batched gradient descent type iteration for the variable θ, one then has to solve a problem of the form Equation (4.3) in every step, which can again be approximated via gradient ascent on the input variable x.

Lipschitz Training of Neural Networks Adversarial robustness is closely related to the Lipschitzness of a network f_θ. Namely, for inputs $x, \bar{x} \in \mathcal{X}$ that are close, we also want the outputs of f_θ to be close. In other words we want to find a small constant $L > 0$ such that

$$\|f_\theta(x) - f_\theta(\bar{x})\|_{\mathcal{Y}} \leq L \ \|x - \bar{x}\|_{\mathcal{X}},$$

where we typically employ an L^p norm for both $\|\cdot\|_{\mathcal{X}}$ and $\|\cdot\|_{\mathcal{Y}}$. The smallest constant fulfilling this inequality is the Lipschitz constant $\mathrm{Lip}(f_\theta)$. If this constant is small, one can expect that small input perturbations do not affect the classification output significantly.

Remark 4.6. Apart from adversarial robustness, having an upper bound on the Lipschitz constant of a neural network is also important for other applications, see, e.g., [Arn+23; Has+20; ACB17]. △

In our work, we employ the Lipschitz constant as a regularizer and consider the problem

$$\min_\theta \mathcal{L}(\theta) + \lambda \ \mathrm{Lip}(f_\theta) \tag{4.4}$$

for a parameter $\lambda > 0$. Computing the Lipschitz constant of neural networks is an NP-hard problem [SV18], therefore many works employ the estimate

$$\mathrm{Lip}(f_\theta) \ \leq \ \prod_{l=1}^{L} \mathrm{Lip}(\Phi_l) \leq C_\sigma \prod_{l=1}^{L} \|W_l\|, \tag{4.5}$$

where here $\|\cdot\|$ denotes some matrix norm and $C_\sigma > 0$ depends on the Lipschitz constants of the activation functions, see [ALG19; Gou+20; KMP20; RKH19]. This inequality is not sharp and usually overestimates the Lipschitz constant, as we see in the following example taken from [CLIP].

Example 4.7. We consider a feed-forward neural network $f_\theta : \mathbb{R} \to \mathbb{R}$ with one hidden layer,

$$\begin{aligned}
\Phi_1(z) &:= \mathrm{ReLU}(W_1 z) := (\max\{z, 0\}, \max\{-z, 0\})^T, \qquad W_1 := (1, -1), \\
\Phi_2(z) &:= W_2 z := z_1 + z_2, \qquad W_2 := (1, 1)^T, \\
f_\theta &:= \Phi_2 \circ \Phi_1.
\end{aligned}$$

For $x \in \mathbb{R}$ we have that

$$\begin{aligned}
x \geq 0 \quad &\Rightarrow \Phi_1(x) = (x, 0)^T \quad \Rightarrow f_\theta(x) = x, \\
x \leq 0 \quad &\Rightarrow \Phi_1(x) = (0, -x)^T \quad \Rightarrow f_\theta(x) = -x,
\end{aligned}$$

and therefore $f_\theta = |\cdot|$, which yields that $\mathrm{Lip}(f_\theta) = 1$. However employing the spectral norm for the weight matrices, we see that

$$\|W_1\| \cdot \|W_2\| = \sqrt{2}\ \sqrt{2} = 2.$$

Plugging the estimate of Eq. (4.5) into Eq. (4.4) therefore potentially over regularizes the problem. While this increases the stability of the network, it can worsen its classification performance.

Contribution in [CLIP] In [CLIP] we propose a strategy to solve Eq. (4.4) approximately, without the estimate in Eq. (4.5). The basic idea consists of approximating the Lipschitz constant on a finite set and using this approximation as a regularizer. Furthermore, we analyse the original model in Eq. (4.4) where we study existence of solutions, the influence of the parameter λ and the limits $\lambda \to 0, \lambda \to \infty$, see Section 4.2.2. Finally, we demonstrate the efficiency of the method by applying it to some simple toy problems, see Section 4.2.3.

4.2.1. Cheap Lipschitz Training

We approximate the Lipschitz constant on a finite set $\mathcal{X}_{\text{Lip}} \subset \mathcal{X} \times \mathcal{X}$ via

$$\text{Lip}(f_\theta; \mathcal{X}_{\text{Lip}}) := \max_{(x,\bar{x})\in\mathcal{X}_{\text{Lip}}} \frac{\|f_\theta(x) - f_\theta(\bar{x})\|_{\mathcal{Y}}}{\|x - \bar{x}\|_{\mathcal{X}}} \approx \text{Lip}(f_\theta).$$

Disregarding the non-differentiable points of $\theta \mapsto \text{Lip}(f_\theta; \mathcal{X}_{\text{Lip}})$ and employing the above approximation, allows us to solve the problem in Eq. (4.4) via stochastic gradient descent on the variable θ.

Remark 4.8. Let $f, g : \mathcal{X} \to \mathcal{Y}$ be two differentiable functions. If $f(\bar{x}) > g(\bar{x})$ for some $\bar{x} \in \mathcal{X}$ then we know that there exists some $\epsilon > 0$ such that $f(x) > g(x)$ for all $x \in B_\epsilon(\bar{x})$. We have that $f \vee g = \max\{f, g\} = f$ in $B_\epsilon(\bar{x})$ and therefore $x \mapsto f(x) \vee g(x)$ is differentiable in $\bar{x}$. If $f(\bar{x}) = g(\bar{x})$ then $f \vee g$ could be non-differentiable at $\bar{x}$. However, in practice we employ automatic differentiation, where in this case one of the functions is chosen, say f, and the *gradient* at $\bar{x}$ is computed as ∇f. △

The strength of this approach is dependent on the quality of the set $\mathcal{X}_{\text{Lip}}$. In [CLIP] we propose to iteratively update the set via a gradient ascent type scheme. Namely, we initialize $\mathcal{X}_{\text{Lip}}$ as a random perturbation of a subset of the given data. In each step we then update the points as follows:

- Consider $L(x, \bar{x}) := \|f_\theta(x) - f_\theta(\bar{x})\| / \|x - \bar{x}\|, \quad x, \bar{x} \in \mathcal{X}$.
- For each $(x, \bar{x}) \in \mathcal{X}_{\text{Lip}}$ perform the update

$$x \leftarrow x + \tau\, L(x, \bar{x}) \nabla_x L(x, \bar{x}),$$
$$\bar{x} \leftarrow \bar{x} + \tau\, L(x, \bar{x}) \nabla_{\bar{x}} L(x, \bar{x}),$$

  for a parameter $\tau > 0$.

This scheme performs gradient ascent on the Lipschitz constant, see [CLIP, Alg. 1]. For each mini-batch $B \subset \mathcal{T}$ we first update the set $\mathcal{X}_{\text{Lip}}$ and then update the parameters via

$$\theta \leftarrow \theta - \eta\, \nabla_\theta \left(\mathcal{L}(\theta; B) + \lambda\ \text{Lip}(f_\theta, \mathcal{X}_{\text{Lip}})\right)$$

which yields the algorithm presented in [CLIP, Alg. 2]. Similar to [Sha+19] one can reuse the gradients computed for ∇_x for the computation of ∇_θ. This fact yields the attribute *cheap* in the Lipschitz training algorithm.

4.2.2. Analysis of Lipschitz Regularization

In [CLIP] we analyse some basic properties of the original regularization problem in Eq. (4.4). We repeat the assumptions made therein.

Assumption 1. We assume that the loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R} \cup \{\infty\}$ satisfies:

a) $\ell(y, \bar{y}) \geq 0$ for all $y, \bar{y} \in \mathcal{Y}$,

b) $y \mapsto \ell(y, \bar{y})$ is lower semi-continuous for all $\bar{y} \in \mathcal{Y}$.

Assumption 2. We assume that the map $\theta \mapsto f_\theta(x)$ is continuous for all $x \in \mathcal{X}$.

Assumption 3. We assume that there exists $\theta \in \Theta$ such that

$$\frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \ell(f_\theta(x), y) + \lambda \operatorname{Lip}(f_\theta) < \infty.$$

Employing only Assumption 2 one can show that that the map $\theta \mapsto \operatorname{Lip}(f_\theta)$ is lower semi-continuous, [CLIP, Lem. 1].

Existence If Θ is compact or finite, one can show that there exist solutions of the problem in Eq. (4.4). In the general case, one needs to add a norm term, that ensures boundedness of a minimizing sequence. The following is the main existence result, which can be proven by the direct method in the calculus of variations [Dac07].

Proposition 4.1 ([CLIP, Prop. 1]). Under Assumptions 1 to 3 the problem

$$\min_{\theta\in\Theta} \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \ell(f_\theta(x), y) + \lambda \operatorname{Lip}(f_\theta) + \mu \|\theta\|_\Theta$$

has a solution for all values $\lambda, \mu > 0$. Here, $\|\cdot\|_\Theta$ denotes a norm on Θ.

Dependency on the Regularization Parameter Assuming that for every $\lambda > 0$ we have a solution $\theta_\lambda \in \Theta$ of Eq. (4.4) one can show that

$$\begin{aligned} \lambda &\longmapsto \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \ell(f_{\theta_\lambda}(x), y) && \text{is non-decreasing,} \\ \lambda &\longmapsto \operatorname{Lip}(f_{\theta_\lambda}) && \text{is non-increasing.} \end{aligned}$$

This statement is the content of [CLIP, Prop. 2], however, the argument is exactly the same as in [BO13]. The intuition behind this result is that with increasing parameter λ, solutions of Eq. (4.4)—more precisely the corresponding networks—have smaller Lipschitz constants, i.e., tend to be more constant, which however also diminishes their expressivity. We formalize the limit cases in the following.

Assuming the realizability condition of [SB14] we know that there exists parameters $\theta \in \Theta$ such that $\mathcal{L}(\theta) = 0$. This means there are parameters that fit the data perfectly. Considering the limit $\lambda \to 0$ we obtain convergence to a solution of the unregularized problem with the smallest Lipschitz constant.

Proposition 4.2 ([CLIP, Prop. 3]). Let Assumptions 1 to 3 and the realizability assumption [SB14] be satisfied. If $\theta_\lambda \to \theta^\dagger \in \Theta$ as $\lambda \searrow 0$, then

$$\theta^\dagger \in \operatorname{argmin} \left\{ \operatorname{Lip}(f_\theta) \,:\, \theta \in \Theta,\, \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \ell(f_\theta(x), y) = 0 \right\}$$

if this problem admits a solution with $\operatorname{Lip}(f_\theta) < \infty$.

Furthermore, we study the effect of sending $\lambda \to \infty$, where we see that the network f_{θ_λ} indeed tends to be constant as expected. We can explicitly characterize this constant as the closest point to barycenter of the output data, that is realizable by a neural network.

Proposition 4.3 ([CLIP, Prop. 4]). Let Assumptions 1 to 3 be satisfied and assume that

$$\mathcal{M} := \{y \in \mathcal{Y} \,:\, \exists \theta \in \Theta,\, f_\theta(x) = y,\, \forall x \in \mathcal{X}\} \neq \emptyset.$$

If $\theta_\lambda \to \theta_\infty \in \Theta$ as $\lambda \to \infty$, then $f_{\theta_\infty}(x) = \hat{y}$ for all $x \in \mathcal{X}$ where

$$\hat{y} \in \operatorname{argmin}_{y'\in\mathcal{M}} \frac{1}{|\mathcal{T}|} \sum_{y\in\mathcal{T}_\mathcal{Y}} \ell(y', y).$$

4.2.3. Numerical Results

We briefly comment on the numerical results [CLIP]. All the experiments were implemented in `Python` [VD95] employing—among others—the `PyTorch` [Pas+19] package. We conduct experiments on the MNIST [LC10] and FashionMNIST [XRV17] datasets.

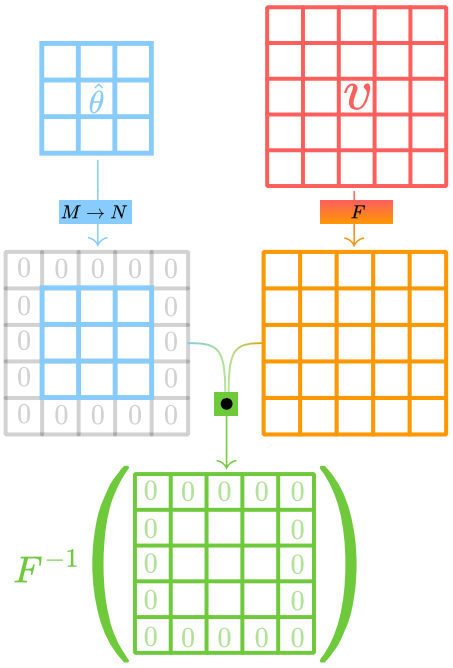
The code for all the experiments is available at github.com/TimRoith/CLIP.

Qualitative Example We first apply the CLIP algorithm to regularize a one-dimensional regression problem. In [CLIP, Fig. 2] one observes the qualitative effect of the regularization. Namely, we are given noisy data from a ground truth function. The solution of the unregularized problem overfits the data and the resulting network has fluctuations and therefore a large Lipschitz constant in certain regions, where the ground truth function has a small Lipschitz constant. With increasing parameter λ, we observe that the networks f_{θ_λ} are smoothed out and better approximate the ground truth function.

Quantitative Evaluation of Adversarial Robustness We additionally evaluate how robust a CLIP trained network is against adversarial attacks. Next to a standard learning rate scheduler, we also update the regularization parameter λ throughout the training process. Namely, we choose a target accuracy and evaluate the loss on a validation set after each epoch. If this accuracy is less than the target, we decrease λ and increase it if the accuracy is higher. In [CLIP, Tab. 1] we compare ourselves against a weight regularization scheme that is based on the estimate in Eq. (4.5). We see that CLIP outperforms this method in most cases.

4.3. Resolution Stability via FNOs: [FNO]

In this section, we comment on the results in [FNO], concerning input robustness w.r.t. resolution changes. In practice we frequently employ the set $\mathcal{X} = [0,1]^{K\times N\times M}$ to represent images. However, from a modeling point of view it is more natural to assume that images are functions $u : \Omega \to [0,1]^K$ where $\Omega \subset \mathbb{R}^2$ is some domain. In this sense, the space $\mathcal{X}$ only constitutes a discretization of the space of all functions from Ω to $[0,1]^K$. The number of pixels, i.e., $N \cdot M$, relates to the *resolution* of the image, where a higher number of pixels yields a higher resolution. In this sense, the resolution is merely an artifact of practical restrictions and should not be relevant for the classification. Therefore, one wants to obtain a resolution-independent classifier, which we study in the following.



Images, Resolution and Scale In the continuum setting we consider the d-dimensional torus $\Omega = \mathbb{R}^d/\mathbb{Z}^d$. An image is a function

$$u : \Omega \to [0,1]^K$$

where $K \in \mathbb{N}$ denotes the number of color channels, see [GW87]. In order to represent images on a computer, we discretize the domain Ω, via a regular grid $\Omega_N = \{x_0, \dots, x_N\}$ indexed by the set

$$\mathcal{J}_N := \{0, \dots, N-1\}^d$$

with grid points $x_j = j/(N-1)$, see e.g. [Kab22; KLM21]. For simplicity, we assume an equal number of discretization points in each dimension. Therefore, N^d is the resolution of an image discretized w.r.t. $\mathcal{J}_N$.

It is important to notice the differences to the concept of *scale*. A change in scale would be a change in the original image domain, for example, by zooming in on a certain area. In our scenario, we usually assume that the image u contains a certain entity to be classified. The scale roughly describes how "big" the entity is, or what percentage of the image it fills. We assume that the scale is fixed. Therefore, changing N directly influences the resolution of the discretization.

How do Neural Networks react to Resolution Changes? In many machine learning applications, one assumes a fixed input size and therefore a fixed resolution, assuming the same scale throughout the data. Formally, networks are defined as mappings $f_\theta : \mathbb{R}^{K\times N\times N} \to \Delta^C$, i.e. they only accept inputs of the fixed resolution N. In order to understand how this constraint can be weakened, we need to consider the concrete structure of our networks, which was similarly done in [KLM21; Kab22]. The architectures we consider are feed-forward networks of the form

$$f_\theta = \Phi^{\text{class}} \circ \mathcal{S} \circ \Phi^{\text{feature}}$$

where

- Φ^{feature} is the so-called feature extractor, which should be applicable independently from the input dimension,
- $\mathcal{S}$ is a function that maps inputs of any size to a fixed output dimension $\mathbb{R}^s$,
- $\Phi^{\text{class}} : \mathbb{R}^s \to \Delta^C$ denotes the classification layer.

The feature extractor usually consists of convolutional layers. As in [Kab22, Ch. 2] we consider the discrete convolution of two discretized images u_N, v_N defined as

$$(u_N * v_N)(x_j) := \sum_{k\in\mathcal{J}_N} u_N(x_k)\cdot v_N(x_{j-k}) \tag{4.6}$$

where for negative indices $j-k$ we set $x_{j-k} := x_{j-k+N}$. This assumes that both u_N and v_N live on the same discretization $\mathcal{J}_N$. For neural networks, we are interested in the convolution of an input image u and a kernel $\theta \in \mathbb{R}^{\mathcal{J}_M}$. Modeling spatial locality—and therefore a small support of the kernel—one usually chooses M much smaller than the resolution. This is for example motivated by the study in [HW62] which explores a similar methodology for the visual cortex of cats. However, if $M < N$ one can not directly employ the convolution in Eq. (4.6), since there we assumed the same discretization for both inputs. In order to account for this dimension mismatch, we consider so-called *spatial zero-padding* for kernels $\theta \in \mathbb{R}^{\mathcal{J}_M}$

$$\theta_k^{M\to N} = \begin{cases} \theta_k & \text{for } k \in \mathcal{J}_N \cap \mathcal{J}_M, \\ 0 & \text{for } k \in \mathcal{J}_N \backslash \mathcal{J}_M. \end{cases}$$

Using this method, one can define the convolution for inputs u_N of arbitrary input resolution $N \geq M$ via

$$C(\theta)(u_N) = \theta^{M\to N} * u_N.$$

In [FNO] we refer to this as the *spatial implementation* of convolution. Up to the behavior on the boundary, this is in fact the standard implementation in most libraries, especially in `PyTorch`. Therefore, a feature extractor consisting of convolutional layers can take inputs of variable resolution. In fact, ignoring possible resolution changes within the

extractor—via pooling or strided convolutions—we have that $\Phi^{\text{feature}}(u_N) \in \mathbb{R}^{\mathcal{J}}_N$ for any $N \in \mathbb{N}$.

The mapping $\mathcal{S}$ can be realized as an adaptive pooling layer, see [Pas+19], which ensures a fixed output size. This methodology yields a *discretization invariant architecture*, see [Kab22; KLM21; Li+20]. This means that from a technical point of view, the network is able to produce outputs for inputs with arbitrary discretization. However, we are actually interested in a *discretization invariant functionality* (see [Kab22; KLM21; Li+20]), which also requires that the output similar for different resolutions.

Input Interpolation The technical possibility to handle different input sizes, as described above, usually does not perform well in practice. This is due to the fact that in the standard spatial implementation of convolution the support of the kernel changes with varying input dimension, hence the output differs, see Fig. 4.1. If a network is trained on a fixed input size, the filters are adapted to this size and therefore only create meaningful responses on this size.

A simple attempt to create a network, such that not only the architecture, but also the functionality is discretization independent—at least up to a certain degree—is input interpolation. In this case, our architecture is modified to

$$\tilde{f}_\theta = f_\theta \circ I$$

where $I : \bigcup_{M \in \mathbb{N}} \mathbb{R}^{\mathcal{J}_M} \to \mathbb{R}^{\mathcal{J}_N}$ is an interpolation function, that maps inputs of arbitrary sizes to a fixed discretization $\mathcal{I}_N$. Typical choices here include nearest neighbor, bilinear or bicubic interpolation, see, e.g., [GW87]. Especially relevant in our case, is so-called *trigonometric interpolation*, where for $v \in \mathbb{R}^{\mathcal{J}_M}$ we define

$$I^{\text{trigo}}(v) := v^{M \xrightarrow{\Delta} N} := F^{-1}\left((Fv)^{M \to N}\right).$$

Contribution in [FNO] We study the connection between FNOs and CNNs for classification problems. We identify under which assumption the architectures are equivalent (see Lemma 4.9), but also where they are not, see Fig. 4.1. Here, we are especially interested in the multi-resolution case. We show that one layer of an FNO is equivariant with respect to trigonometric interpolation, Corollary 4.11. This is also underlined by numerical experiments, where we compare the FNO implementation to interpolation methods and a simple CNN implementation. Furthermore, we show that training equivalent CNN and FNO layers leads to different results, i.e., while they have the same forward pass, the gradients w.r.t. their parameters might differ, Lemma 4.10. Moreover, we show continuity and Fréchet-differentiability of abstract neural layers as operators between L^p spaces, see Section 4.3.2. Finally, we conduct numerical experiments supporting our theoretical findings Section 4.3.3.

4.3.1. Fourier Neural Operators

We want to obtain neural networks whose output does not depend on the discretization of the image $u : \Omega \to \mathbb{R}^K$. This raises the question whether it is possible to find a

formulation that allows us to work in the infinite dimensional setting. In [Kov+21] this issue was addressed in the setting of parametric PDEs, where the authors introduced the concept of *neural operators*. One layer of a neural operator is given as a mapping $\mathcal{G} : L^p(\Omega) \to L^q(\Omega)$

$$\mathcal{G}(u)(x) = \sigma\left(\Psi(u)(x)\right) \qquad \text{for a.e. } x \in \Omega, \tag{4.7}$$

with an affine linear part given by

$$\Psi(u) = Wu + \mathcal{K}u + b, \tag{4.8}$$

where

- $\mathcal{K} : u \mapsto \int_\Omega \kappa(\cdot, y)\, u(y)\, dy$ is a kernel integral operator with kernel $\kappa : \Omega \times \Omega \to \mathbb{R}$,
- $W \in \mathbb{R}$ models a residual component,
- and $b : \Omega \to \mathbb{R}$ models a bias.

By a slight abuse of notation, the activation function $\sigma : \mathbb{R} \to \mathbb{R}$ acts as a Nemytskii operator, i.e.,

$$\sigma : v \mapsto \sigma(v(\cdot)), \tag{4.9}$$

see, e.g., [Trö10]. In Section 4.3.2 we analyze continuity and differentiablility of a layer in this abstract form. However, the most relevant case for us, is when $\mathcal{K}$ is a convolution operator, i.e., $\kappa(x, y) = \kappa(x - y)$ is a translation invariant kernel. In this special case $\mathcal{G}$ is then known as layer of a *Fourier Neural Operator* (FNO) as introduced in [Li+20]. We can parameterize the kernel via its Fourier coefficients $\hat{\theta}_k \in \mathbb{C}$

$$\kappa_{\hat{\theta}}(x) = \sum_{k \in \mathcal{I}} \hat{\theta}_k\, b_k(x), \tag{4.10}$$

where $b_k(x) = \exp\left(2\pi i\, kx\right)$ denote the Fourier basis functions. In practice, we assume that $\kappa_{\hat{\theta}}$ only has a finite amount of non-zero Fourier coefficients. We choose the set $\mathcal{I}_N := \{-\lceil (N-1)/2 \rceil, \dots, 0, \dots, \lfloor (N-1)/2 \rfloor\}^d$ as the index set, as done in [Li+20]. Employing this set with odd $N \in \mathbb{N}$, we can easily enforce Hermitian symmetry $\hat{\theta}_k = \overline{\hat{\theta}_{-k}}$, which ensures that $\mathcal{K}_{\hat{\theta}}$ outputs real-valued functions. For now, we assume an odd number N and deal with the even case later. We note that this number of Fourier coefficients is completely independent of the spatial input discretization, which is the main advantage of FNOs.

How to Perform Convolutions with FNOs? In [FNO] we consider the discrete Fourier transform and its inverse

$$(Fv)_k = \frac{1}{\lambda} \sum_{j \in \mathcal{J}_N} v_j\, e^{-2\pi i \left\langle k, \frac{j}{N} \right\rangle}, k \in \mathcal{I}_N,$$

$$\left(F^{-1}\hat{v}\right)_j = \frac{\lambda}{|\mathcal{J}_N|} \sum_{k \in \mathcal{I}_N} \hat{v}_k\, e^{2\pi i \left\langle k, \frac{j}{N} \right\rangle} j \in \mathcal{J}_N,$$

with a normalization constant $\lambda \in \{1, \sqrt{|\mathcal{J}_N|}, |\mathcal{J}_N|\}$. We only consider parameters $\hat{\theta} \in \mathbb{C}^{\mathcal{I}_N}_{\text{sym}} := F(\mathbb{R}^{\mathcal{J}_N})$, where we can employ the convolution theorem (see e.g. [Gra14]) to define

$$K(\hat{\theta})(v) = F^{-1}\left(\hat{\theta} \cdot Fv\right) \qquad \text{for } v \in \mathbb{R}^{\mathcal{J}_N}, \tag{4.11}$$

which is referred to as the FNO or spectral implementation of convolution.

How do FNOs React to Resolution Changes? The number of Fourier coefficients of $\hat{\theta} \in \mathbb{C}^{\mathcal{I}_M}_{\text{sym}}$ is independent of the input resolution. Nevertheless, the point-wise multiplication in Eq. (4.11) is only defined for inputs $v \in \mathbb{R}^{\mathcal{J}_M}$, thus the question arises how FNOs can be adapted to dimension mismatch. Here, we use a conceptually similar idea, by employing zero-padding. However, the important and major difference is that this zero-padding is performed in the spectral domain. Assuming that $N > M$ is odd we define

$$\hat{\theta}^{M\to N}_k = \begin{cases} \hat{\theta}_k & \text{for } k \in \mathcal{I}_N \cap \mathcal{I}_M, \\ 0 & \text{for } k \in \mathcal{I}_N \backslash \mathcal{I}_M, \end{cases}$$

and the spectral implementation of convolution for $v_N \in \mathbb{R}^{\mathcal{J}_N}$ is given as

$$K(\hat{\theta})(v_N) := K(\hat{\theta}^{M\to N})(v_N).$$

What Is The Difference Between Standard and FNO Implementation? The difference between the spectral and spatial implementation is best explained in [FNO, Fig. 1], which we repeat in Fig. 4.1 for convenience. We are given a kernel $\theta \in \mathbb{R}^{\mathcal{J}_M}$, its unnormalized Fourier transform $\hat{\theta} = \lambda F(\theta)$ and an input $v_N \in \mathbb{R}^{\mathcal{J}_N}$. If $M = N$, i.e., all dimensions match, we observe that spectral and spatial implementation are equivalent, see the middle row of Fig. 4.1. However, if we consider a higher resolution variant of the image with $N > M$, spatial zero-padding results in the kernel being localized in space and therefore the effect of convolving it with v_N changes. On the other hand for the spectral implementation we observe an equivariant behaviour. The resolution of the output changes but qualitatively the effect of the filter stays the same.

Connection to Interpolation As hinted in Fig. 4.1, when changing the resolution, the spectral implementation of convolution can be interpreted as a standard convolution with an interpolated kernel. In fact we observe that for $\theta \in \mathbb{R}^{\mathcal{J}_M}, \hat{\theta} = \lambda F(\theta)$ and $v_N \in \mathbb{R}^{\mathcal{J}_N}$ we have that

$$\begin{aligned} K(\hat{\theta})(v_N) &= F^{-1}\left(\hat{\theta}^{M\to N} \cdot Fv_N\right) \\ &= F^{-1}\left(F\,F^{-1}\hat{\theta}^{M\to N} \cdot Fv_N\right) \\ &= F^{-1}\left(F\theta^{M\xrightarrow{\Delta}N} \cdot Fv_N\right) = \theta^{M\xrightarrow{\Delta}N} * v_N \\ &= C(\theta^{M\xrightarrow{\Delta}N})(v_N). \end{aligned}$$

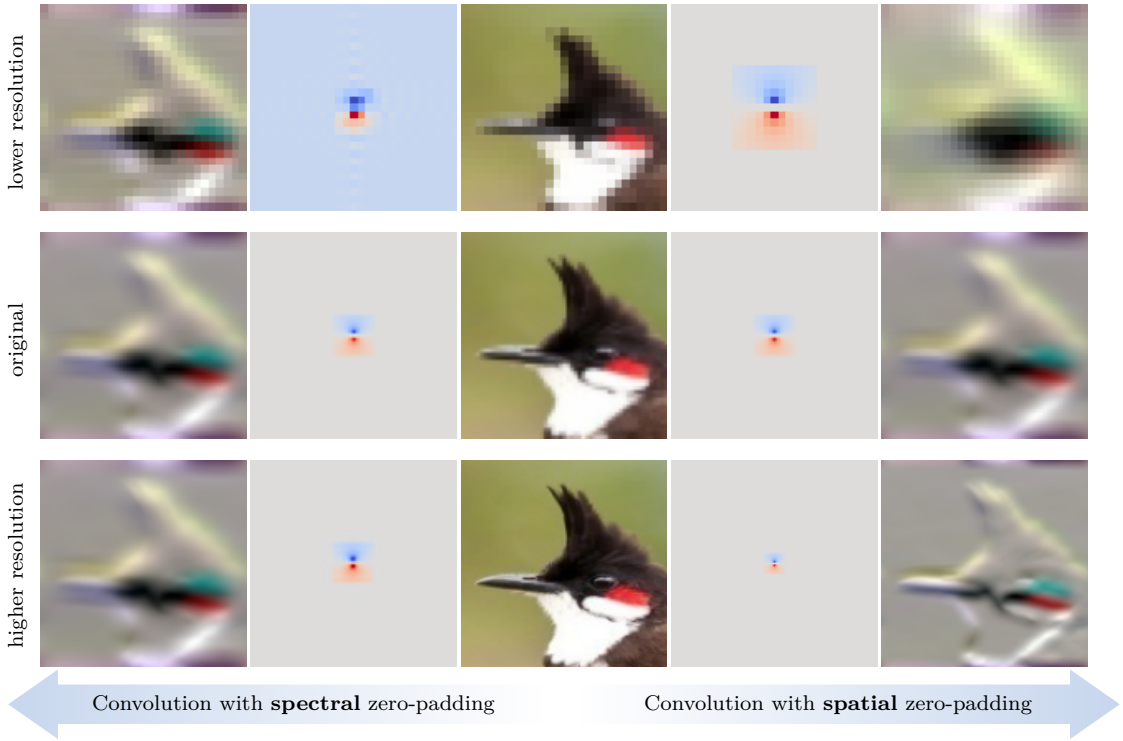


Figure 4.1.: The image is taken from [Kab22, Fig. 1]—depicting a red whiskered bulbul taken from the Birds500 dataset [Pio21]—and visualizes the different effects of spectral and spatial zero-padding.

Therefore, applying one FNO layer is equivalent to applying a standard CNN layer with a trigonometric interpolation of the kernel.

Adaption to Even Dimensions Zero-padding of the spectral coefficients only fulfills Hermitian symmetry in the case where M, N are odd. In order to adapt this to the even case, we employ so-called Nyquist splitting, see [Bri19]. In all our experiments, the implementation carefully employs this method, which ensures that the output of the spectral convolution is real valued. We also refer to [FNO, Sec. 3.3], where the details on this topic are given.

4.3.2. Analytical Results for FNOs

In this section, we comment on the theoretical findings in [FNO]. We first consider the abstract neural layer as in Eq. (4.7) for which we show continuity and Fréchet-differentiability.

Continuity of Neural Layers The results for neural layers in [FNO] mostly fall back to the theory of Nemytskii operators, see, e.g., [Trö10; AP93]. In order to show that the

layer $\mathcal{G}$ is a well defined mapping from L^p to L^q one first needs to identify an exponent $r \in [1, \infty]$ such that the affine part is a mapping $\Psi : L^p \to L^r$. For example if $\kappa \in L^s$ with $1/r + 1 = 1/p + 1/s$ it follows from Young's convolution inequality that $\mathcal{K}$ maps to L^r, see [Gra14, Th. 1.2.12]. If then $W = 0$ and $b \in L^r$ we know that Ψ maps to L^r.

To ensure that the Nemytskii operator defines a mapping $\sigma : L^r \to L^q$ one needs to assume a growth condition on $\sigma : \mathbb{R} \to \mathbb{R}$, see [FNO, Eq. 4] and originally [Trö10]. Under these assumptions, we obtain [FNO, Prop. 1]. Concrete examples fulfilling these assumptions are given in [FNO] borrowing concepts from [AP93; Trö10]. The most important activation function that is valid in this setting is the ReLU function

$$\mathrm{ReLU}(x) = \max\{x, 0\}.$$

Proposition 4.4 ([FNO, Prop. 1]). For $1 \leq p, q \leq \infty$ let $\mathcal{L}$ be an operator layer given by (4.7) with an activation function $\sigma : \mathbb{R} \to \mathbb{R}$. If there exists $r \geq 1$ such that

(i) the affine part defines a mapping $\Psi : L^p(\Omega) \to L^r(\Omega)$,

(ii) the activation function σ generates a Nemytskii operator $\sigma : L^r(\Omega) \to L^q(\Omega)$,

then it holds that $\mathcal{L} : L^p(\Omega) \to L^q(\Omega)$. If additionally Ψ is a continuous operator on the specified spaces and the function σ is continuous, or uniformly continuous in the case $q = \infty$, the operator $\mathcal{L} : L^p(\Omega) \to L^q(\Omega)$ is also continuous.

Differentiability of Neural Layers We furthermore consider Fréchet differentiability of a neural layer w.r.t. the input variable. This can also be transferred to differentiability w.r.t. the parameters as we show in [FNO, Ex. 4]. Conceptually, the main result we repeat here is similar to the one on continuity of the last paragraph. The major difference is that we also need to assume differentiability of the activation function, also assuming a growth condition on its derivative. The ReLU function can therefore not be chosen in this setting, however the smooth approximation called GELU ([HG16])

$$\mathrm{GELU}(x) := x\,\Phi(x)$$

where Φ is the CDF of the standard normal distribution, can be employed.

Proposition 4.5 ([FNO, Prop. 2]). For $1 \leq p, q \leq \infty$, let $\mathcal{L}$ be an operator layer given by (4.7) with affine part Ψ as in (4.8). If there exists $r > q$, or $r = q = \infty$ such that

(i) the affine part is a continuous operator $\Psi : L^p(\Omega) \to L^r(\Omega)$,

(ii) the activation function $\sigma : \mathbb{R} \to \mathbb{R}$ is continuously differentiable

(iii) and the derivative of the activation function generates a Nemytskii operator $\sigma' : L^r(\Omega) \to [L^r(\Omega) \to L^s(\Omega)]$ with $s = rq/(r - q)$,

then it holds that $\mathcal{L} : L^p(\Omega) \to L^q(\Omega)$ is Fréchet-differentiable in any $v \in L^p(\Omega)$ with Fréchet-derivative $D\mathcal{L}(v) : L^p(\Omega) \to L^q(\Omega)$

$$D\mathcal{L}(v)(h) = \sigma'(\Psi(v)) \cdot \tilde{\Psi}(h),$$

where $\tilde{\Psi}$ denotes the linear part of Ψ, i.e., $\tilde{\Psi} = \Psi - b$.

Convertibility Between FNOs and CNNs The following lemma formalizes the intuition that FNOs and CNNs are equivalent in certain settings. Given inputs of a fixed discretization $\mathcal{J}_N$ and parameters $\theta \in \mathbb{R}^{\mathcal{J}_M}$, the standard convolution implementation is equivalent to the spectral one w.r.t. the parameters $\hat{\theta} = \lambda\, F(\theta^{M \to N})$. The subtle but important point here, is that the number of spectral parameters needs to be equal to the input size in order to achieve equivalence. In [FNO, Fig.3] we observe numerically that the number of used spectral coefficients actually needs to match the input size in order to achieve equivalence.

Lemma 4.9 ([FNO, Lem. 3]). Let $M \leq N$ both be odd and let $T : \mathbb{R}^{J_N} \to \mathbb{C}^{I_N}$ be defined for $\theta \in \mathbb{R}^{J_N}$ as $T(\theta) = \lambda\, F(\theta)$. For any $\theta \in \mathbb{R}^{J_M}$ and $v \in \mathbb{R}^{J_N}$ it holds true that

$$C(\theta)(v) = K(T(\theta^{M \to N}))(v)$$

and for any $\hat{\theta} \in \mathbb{C}^{I_M}_{\text{sym}}$ and $v \in \mathbb{R}^{J_N}$ it holds true that

$$K(\hat{\theta})(v) = C(T^{-1}(\hat{\theta}^{M \to N}))(v).$$

Together with [FNO, Fig. 3] we see that in order to convert a CNN to a FNO we need a large number of spectral parameters. This is also connected to the fact that spatial locality can only be expressed using more spectral coefficients. Therefore, one might think that FNOs are infeasible due to memory requirements. However, it turns out that directly optimizing over the spectral parameters leads to a comparable performance, already for a low number of Fourier coefficients, which is reported in the blue curve in [FNO, Fig. 3]. This hints that training a FNO via gradient descent leads to different weights, that are not simply a conversion of a similarly trained CNN. The reason for these differences is that the gradients of spectral and standard convolutional layers, that have the same forward pass, are not directly convertible via the Fourier transformation. In fact, one obtains an additional factor, which we formalize in the following lemma.

Lemma 4.10 ([FNO, Lem. 4]). For odd $N \in \mathbb{N}$ and $v, \theta \in \mathbb{R}^{J_N}$ and $\hat{\theta} = T(\theta)$ it holds true that

$$\nabla_{\hat{\theta}} K(\hat{\theta})(v) = \frac{1}{|J_N|}\, T\left(\nabla_\theta C(\theta)(v)\right).$$

Interpolation Equivariance The last result considers the main motivation of the whole section, namely, the resolution invariance of an FNO layer. However, we can not allow an arbitrary resizing operation of the input. Employing the spectral implementation of convolution layer, we can show equivariance w.r.t. trigonometric interpolation of the input.

Corollary 4.11. For $\hat{\theta} \in \mathbb{C}^{I_M}_{\text{sym}}$, $v \in \mathbb{R}^{J_N}$, $M \leq N$ it holds true for any $L \geq M$ that

$$K(\hat{\theta})(v^{N \xrightarrow{\Delta} L}) = \Big(K(\hat{\theta})(v)\Big)^{N \xrightarrow{\Delta} L}.$$

4.3.3. Numerical Results

In the numerical section of [FNO] we study the convertibility of CNNs to FNOs as discussed in Section 4.3.2 and the resolution invariance of the different proposed approaches. Again we employ—among others—the `PyTorch` package [Pas+19]. The experiments are conducted on the FashionMNIST [XRV17] and a former version of the BIRDS500 [Pio21] dataset. We remark that our implementation carefully treats the case of even kernel or input sizes, via Nyquist splitting. This allows us to apply all the derived results for the odd and also the even case.

The code for all the experiments is available at https://github.com/samirak98/FourierImaging.

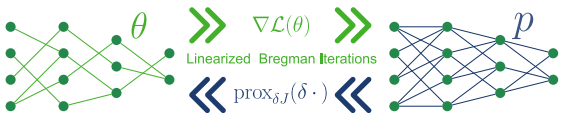
Convertibility and Training Differences As already described in Section 4.3.2, the first experiment, displayed in [FNO, Fig. 3], studies how CNNs can be converted to FNOs. Here, we employ a network with two convolutional layers for feature extraction and a linear classification layer. We train this network—in the spatial implementation—on the FashionMNIST dataset [XRV17] employing varying spatial kernel sizes. Here, we see that for $M = 5$ the spatial implementation already has the best performance and adding more parameters does not increase the performance. We then convert a set of spatial parameters to the Fourier formulation, employing again a varying number of spectral coefficients. It turns out that the requirement of [FNO, Lem. 3], that the number of coefficients must match the input dimension, is indeed relevant. We only obtain the full performance of the CNN if we use all spectral parameters. However, the example also visualizes [FNO, Lem. 4], namely that training a FNO conceptually leads to a different set of parameters. Optimizing over the spectral parameters yields a comparable performance already for a smaller number of coefficients.

Resolution Invariance In the second experiment we study the resolution invariance of the three discretization independent architectures we considered, namely the naive CNN adaptation, input interpolation and the FNO implementation. We first train the convolutional architecture as in the previous paragraph on the FashionMNIST dataset, that has an input size of 28×28. We resize the input data via trigonometric and bilinear interpolation—referred to as data sizing—to simulate multi-resolution data. One expects that the classification performance drops when the data is resized to a smaller size, since this step looses information. However, resizing the images to higher resolutions should yield the same performance.

In [FNO, Fig. 4] we see that the simple adaption does not perform well both in the lower and the higher resolution setting. Employing input interpolation improves the performance in the lower resolution setting. In the higher resolution regime, we obtain a constant performance, which is expected. Finally, we see that the FNO—which was converted from the CNN—performs as well as input interpolation, which justifies the resolution independence of this architecture.

In the second experiment, we trained a ResNet18 [He+16a] on a former version of the BIRDS500 dataset [Pio21], with an input size of 112×112. Concerning the naive adaption and the input interpolation, we observe the same behavior as in the previous example. However, the FNO variant performs slightly worse, which is surprising, especially in the higher resolution regime. In [FNO] we conclude that this is due the dimension changes within the ResNet architectures. These changes occur due to strided convolutions or pooling operations between the layers. In fact, in order to achieve the performance as displayed in [FNO, Fig. 4 (b)] we replaced every striding by a trigonometric interpolation, which fits better into the FNO framework and vastly improves the performance. Therefore, we summarize that architecture intern dimension changes—which are very common in practice—can potentially hinder resolution invariance.

4.4. Sparsity via Bregman Iterations: [BREG-I]



In this section, we now consider parameter sparsity of neural networks. Starting from the first simple architectures (e.g. [Ros58]) the size of typical networks has grown significantly in the last years [Hoe+21]. While this development allowed to solve increasingly difficult tasks with neural networks, it also lead to immense computational requirements, both for the training and evaluation of the net. Here, the question arises, how these computational tasks can be made more efficient. We list some of the approaches, below and refer to [Gho+21] for a comprehensive overview.

- **Architecture Optimization**: Designing an architecture that can be trained to have the same performance as a comparable one, while having less parameters, e.g. [EMH19; How+17].
- **Quantization**: Lowering the precision of the machine numbers of the parameters, e.g., [Ban+18; CBD14].
- **Knowledge distillation**: Employing a trained larger network to learn a smaller one, in a student-teacher approach, e.g., [Sch92; HVD15].
- **Pruning**: Parameters with small saliency, i.e., parameters that contribute little to the effective output of the network are removed or set to zero, in order to obtain a sparse weight matrix or a smaller architecture, see, e.g., [LDS89; HSW93].

From the above methods, the pruning approach is most influential to our work in [BREG-I]. Namely, the concept of compressing the neural network by sparsifying the weight matrices is similarly employed. However, there are a few deviations from the classical pruning framework, which we note in the following.

Sparsity via Optimization In [LDS89; HSW93] one assumes to be given a trained neural network, which is then compressed after training based on some criterion. The authors in [CFP97] employ an iterative pruning scheme, where a similar criterion is employed, in order to throw away certain weights after each step. In our work we want to employ a L^1 type penalty (see [CM73]) on the weight matrices $W \in \mathbb{R}^{n\times m}$, i.e.,

$$\|W\|_1 = \sum_{i=1}^{n}\sum_{j=1}^{m} |W_{ij}|\,.$$

In [Tib96] this leads to the so-called Lasso problem

$$\min_{\theta} \mathcal{L}(\theta) + \lambda \|\theta\|_1\,, \qquad \lambda > 0,$$

where we extend the L^1 norm to the parameter space Θ as discussed in Section 4.4.5. This problem can for example be solved by a proximal gradient descent iteration

$$\theta^{(k+1)} = \text{soft shrinkage}(\theta - \tau\nabla\mathcal{L}(\theta^{(k)})) \tag{4.12}$$

where the shrinkage operator is defined in Example 4.18. This iteration was employed in [Nit14; RVV20; Red+16] to train sparse neural networks.

Sparse-to-Sparse Training All of the sparsity-based methods mentioned before, start with dense weight matrices and only decrease the number of parameters during or at the end of the training process. Our approach yields an iteration, where the network is sparse throughout the iteration. In fact we start with only very few non-zero parameters and only activate necessary weights during training. This paradigm is known as sparse-to-sparse or evolutionary training [Moc+18; DZ19; Evc+20; DYJ19; Fu+22; Hua+16; Liu+21].

Contribution in [BREG-I] Our work falls into the regime of sparse-to-sparse training. However, instead of relying on some heuristic growth strategy, we employ the concept of inverse scale flows (see Section 4.4.1) which allows us to obtain a optimization-driven framework, with a time-continuous interpretation. We propose a stochastic variant of linearized Bregman iterations (Section 4.4.3) and employ it to train a sparse neural network. We show monotonic decrease of the loss in the stochastic setting—which is not possible in the case of proximal gradient descent Eq. (4.12)—and convergence of the iterates under additional convexity assumptions, see Section 4.4.4. Finally, we demonstrate the numerical efficiency of the method (see Section 4.4.5) and provide an interesting applications for neural architecture search, which was further developed in [BREG-II].

4.4.1. Preliminaries on Convex Analysis and Bregman Iterations

We first review some necessary concepts from convex analysis that allow us to introduce the framework in [BREG-I]. We refer to [BB18; Roc97; BC11] for a more exhaustive introduction to these topics. In the following we focus on a lower semi-continuous regularization functional $J : \Theta \to (-\infty, \infty]$. Here, J is called lower semi-continuous if $J(u) \leq \liminf_{n\to\infty} J(u_n)$ holds for all sequences $(u_n)_{n\in\mathbb{N}} \subset \Theta$ converging to u. Furthermore, we require the functional to be convex.

Definition 4.12. Given a Hilbert space Θ and a functional $J : \Theta \to (-\infty, \infty]$.

1. The functional J is called convex, if

$$J(\lambda\overline{\theta} + (1-\lambda)\theta) \leq \lambda J(\overline{\theta}) + (1-\lambda)J(\theta), \quad \forall \lambda \in [0,1],\, \overline{\theta}, \theta \in \Theta. \tag{4.13}$$

2. The effective domain of J is defined as $\mathrm{dom}(J) := \{\theta \in \Theta : J(\theta) \neq \infty\}$ and J is called proper if $\mathrm{dom}(J) \neq \emptyset$.

We want to consider functionals J that are convex, but not necessarily differentiable. Therefore, we define the subdifferential.

Definition 4.13. The subdifferential of a convex and proper functional $J : \Theta \to (-\infty, \infty]$ at a point $\theta \in \Theta$ is given as

$$\partial J(\theta) := \left\{ p \in \Theta \,:\, J(\theta) + \langle p, \overline{\theta} - \theta \rangle \leq J(\overline{\theta}),\ \forall \overline{\theta} \in \Theta \right\}. \tag{4.14}$$

If J is differentiable, then the subdifferntial coincides with the classical gradient (or Fréchet derivative). We denote by $\mathrm{dom}(\partial J) := \{\theta \in \Theta \,:\, \partial J(\theta) \neq \emptyset\}$ and observe that $\mathrm{dom}(\partial J) \subset \mathrm{dom}(J)$.

The Bregman Distance The main algorithm in this section are so-called Bregman iterations, which make use of the Bregman distance.

Definition 4.14 (Bregman Distance). Let $J : \Theta \to (-\infty, \infty]$ be a proper and convex functional. Then, for $\theta \in \mathrm{dom}(\partial J), \overline{\theta} \in \Theta$ we define

$$D_J^p(\overline{\theta}, \theta) := J(\overline{\theta}) - J(\theta) - \langle p, \overline{\theta} - \theta \rangle, \quad p \in \partial J(\theta). \tag{4.15}$$

For $p \in \partial J(\theta)$ and $\overline{p} \in \partial J(\overline{\theta})$ we define the *symmetric* Bregman distance as

$$D_J^{\mathrm{sym}}(\overline{\theta}, \theta) := D_J^p(\overline{\theta}, \theta) + D_J^{\overline{p}}(\theta, \overline{\theta}). \tag{4.16}$$

Intuitively, the Bregman distance $D_J^p(\overline{\theta}, \theta)$, measures the distance of J to its linearization around θ, see Fig. 4.2. If J is differentiable, then the subdifferential is single valued—we can suppress the sup script p—and we obtain

$$D_J(\overline{\theta}, \theta) = J(\overline{\theta}) - J(\theta) - \langle \nabla J(\theta), \overline{\theta} - \theta \rangle.$$

Example 4.15. For $\Theta = \mathbb{R}^d$ and $J = \frac{1}{2} \|\cdot\|_2^2$ we see that $\partial J(\theta) = \{\theta\}$ and therefore

$$\begin{aligned} D_J^p(\overline{\theta}, \theta) &= \frac{1}{2}\langle \overline{\theta}, \overline{\theta} \rangle - \frac{1}{2}\langle \theta, \theta \rangle - \langle \theta, \overline{\theta} - \theta \rangle \\ &= \frac{1}{2}\langle \overline{\theta}, \overline{\theta} \rangle + \frac{1}{2}\langle \theta, \theta \rangle - \langle \theta, \overline{\theta} \rangle \\ &= \frac{1}{2} \left\| \overline{\theta} - \theta \right\|_2^2 = J(\overline{\theta} - \theta). \end{aligned}$$

We can easily see, that in general this "distance" is neither definite, symmetric nor fulfills the triangle inequality, hence it is not a metric. However, it fulfills the two distance axioms

$$D_J^p(\overline{\theta}, \theta) \geq 0, \quad D_J^p(\theta, \theta) = 0, \quad \forall \overline{\theta} \in \Theta, \theta \in \mathrm{dom}(\partial J). \tag{4.17}$$

The same holds for the symmetric Bregman distance, where additionally—as the name suggests—the symmetry property is fulfilled.

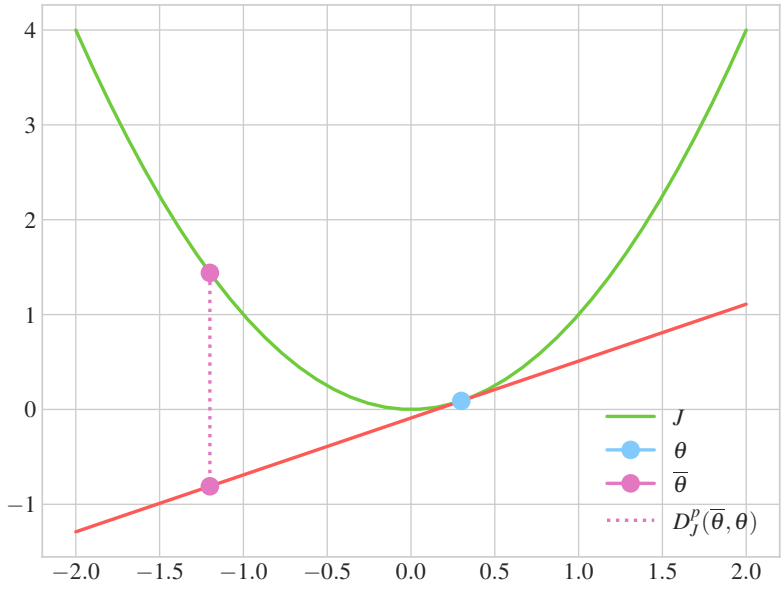


Figure 4.2.: Visualization of the Bregman distance.

The Proximal Operator Another crucial concept in this section, is the so-called proximal operator.

Definition 4.16. Let $J : \Theta \to (-\infty, \infty]$ be convex, proper and lower semicontinuous functional, then we define the *proximal operator* as

$$\operatorname{prox}_J(\bar{\theta}) := \operatorname{argmin}_{\theta \in \Theta} \frac{1}{2} \left\| \theta - \bar{\theta} \right\|^2 + J(\theta).$$

If J is a closed function, i.e., its sublevel sets

$$N_\alpha = \{\theta \in \operatorname{dom} J : J(\theta) \leq \theta\}$$

are closed for every $\alpha \in \mathbb{R}$ then we have that the function $\tilde{J} = \frac{1}{2} \|\theta - \cdot\|^2 + J(\theta)$ is closed, proper and *strongly* convex and therefore has a unique minimizer, see [Roc97, Thm. 27.1]. Additionally, one often considers a regularization parameter $\lambda > 0$ and is then interested in $\operatorname{prox}_{\lambda J}$.

Remark 4.17. The the optimality conditions for $\theta = \operatorname{prox}_{\lambda J}(\bar{\theta})$ yield

$$\bar{\theta} - \theta \in \lambda \, \partial J(\theta).$$

For a proper, closed and convex function we obtain

$$\theta = (I + \lambda \partial J)^{-1}(\bar{\theta})$$

where $(I + \lambda \partial J)^{-1}$ is called the *resolvent* and is a one-to-one mapping (see [PB+14, Ch. 3.2]) which justifies the equality in the above equation. If J is differentiable, then we

have $\partial J = \{\nabla J\}$ and therefore,

$$\operatorname{prox}_{\lambda J} = (I + \lambda \nabla J)^{-1}.$$

△

In the following we list two relevant examples for the applications in [BREG-I; BREG-II].

Example 4.18. If $J = \|\cdot\|$ is a norm and $\lambda > 0$ then we have that (see, e.g., [PB+14])

$$\operatorname{prox}_{\lambda J}(\overline{\theta}) = \overline{\theta} - \operatorname{Proj}_{\|\cdot\|^*}(\overline{\theta}/\lambda)$$

where $\operatorname{Proj}_{\|\cdot\|^*}$ denotes the projection operator w.r.t. the dual norm $\|\theta\|^* = \sup\{|\langle f, \theta\rangle| : f \in \Theta^*\}$. In the case of ℓ^p norms on $\mathbb{R}^d$ we know that that

$$\|\theta\|_p^* = \|\theta\|_q$$

with $1/p + 1/q = 1$ with the notational convention of $1/\infty = 0$. Especially relevant are the cases $p \in \{1, 2\}$. Here, we then have that

$$\operatorname{prox}_{\lambda\|\cdot\|_2}(\overline{\theta}) = \overline{\theta}\left(1 - \min\left\{\frac{\lambda}{\left\|\overline{\theta}\right\|_2}, 1\right\}\right) = \begin{cases} \overline{\theta}\left(1 - \lambda/\left\|\overline{\theta}\right\|_2\right) & \text{if } \left\|\overline{\theta}\right\|_2 \geq \lambda, \\ 0 & \text{else} \end{cases}$$

and for $i = 1, \ldots, n$,

$$\operatorname{prox}_{\lambda\|\cdot\|_1}(\overline{\theta})_i = \operatorname{sign}(\overline{\theta}_i)\ \max\left\{\left|\overline{\theta}_i\right| - \lambda, 0\right\} = \begin{cases} \overline{\theta}_i - \lambda & \text{if } \overline{\theta} > \lambda, \\ 0 & \text{if } \left|\overline{\theta}_i\right| \leq \lambda, \\ \overline{\theta}_i + \lambda & \text{if } \overline{\theta} < -\lambda, \end{cases}$$

the so called *soft shrinkage operator*.

Example 4.19 (Group Norms). Another relevant functional J is the group norm $\ell_{1,2}$ that—in the context of sparse neural networks—was first employed by [Sca+17]. Here, we assume that the parameters in $\theta \in \Theta$ can be grouped in a collection of parameters, i.e., $\theta = \{\theta_1, \ldots, \theta_s\}$, and we choose

$$J(\theta) = \sum_{g \in \theta} \sqrt{s}\ \|g\|_2 .$$

In this case the proximal operator is given as

$$\operatorname{prox}_{\lambda J}(\overline{\theta})_i = \theta_i\ \max\left\{1 - \min\left\{\frac{\lambda\sqrt{s}}{\|\theta\|_2}, 1\right\}, 0\right\}.$$

Bregman Iterations Our goal is to minimize a function $\mathcal{L}$ while simultaneously obtaining a low value w.r.t. the functional J. A popular approach considers the regularized problem

$$\min_\theta \mathcal{L}(\theta) + \lambda J(\theta) \qquad \lambda > 0,$$

see e.g. [Tik43; CP08; DDD04; FS06; FNW07; Cha04; CP11], which however influences the minimizers of the original problem $\min_\theta \mathcal{L}(\theta)$. In the derivation of the Bregman iterations one can take a different viewpoint. We want to employ an iterative scheme, where in each step we minimize $\mathcal{L}$ while penalizing the distance to the previous iterate. For a stepping parameter τ and starting from some $\theta^{(0)} \in \Theta$ this yields the update

$$\theta^{(k+1)} = \operatorname{argmin}_\theta \tau\mathcal{L}(\theta) + \frac{1}{2}\left\|\theta - \theta^{(k)}\right\|^2 = \operatorname{prox}_{\tau\mathcal{L}}(\theta^{(k)}). \tag{4.18}$$

This concept is known as the proximal point algorithm [Bre67] as well as the minimizing movement scheme [De 93]. If $\mathcal{L}$ is differentiable, this update can be rewritten as

$$\theta^{(k+1)} = (I + \tau\nabla\mathcal{L})^{-1}\theta^{(k)} \Leftrightarrow \frac{1}{\tau}\left(\theta^{(k+1)} - \theta^{(k)}\right) = -\nabla\mathcal{L}(\theta^{(k+1)})$$

which is a implicit Euler discretization ([Eul24]) of the time-continuos gradient flow

$$\partial_t\theta_t = -\nabla\mathcal{L}(\theta_t).$$

We see that the penalization term in Eq. (4.18) is in fact the Bregman distance w.r.t. the functional $\frac{1}{2}\|\cdot\|^2$, see Example 4.15. In order to incorporate an arbitrary convex functional J—and therefore allow each iterate to only slightly deviate w.r.t. the Bregman distance of J to the previous iterate—we employ $D_J^{p^{(k)}}(\cdot, \theta^{(k)})$ as a penalization term. Here, we obtain a update scheme for the subgradients $p^{(k)}$, as follows,

$$\theta = \operatorname{argmin}_{\theta\in\Theta} D_J^{p^{(k)}}\left(\theta, \theta^{(k)}\right) + \tau\mathcal{L}(\theta) \tag{4.19}$$

$$\Leftrightarrow p^{(k)} + \tau\nabla\mathcal{L}(\theta) \in \partial J(\theta). \tag{4.20}$$

This yields the *Bregman iteration* of [Osh+05]

$$\theta^{(k+1)} = \operatorname{argmin}_{\theta\in\Theta} D_J^{p^{(k)}}\left(\theta, \theta^{(k)}\right) + \tau\mathcal{L}(\theta), \tag{4.21a}$$

$$p^{(k+1)} = p^{(k)} - \tau\nabla\mathcal{L}(\theta^{(k+1)}) \in \partial J(\theta^{(k+1)}). \tag{4.21b}$$

The nature of Bregman iterations requires starting with an iterate $\theta^{(0)}$ that has a low value in J—preferably $J(\theta^{(0)}) = 0$—and only increase $J(\theta^{(k)})$ gradually in each step.

Remark 4.20. Originally, the iterations were employed for solving inverse problems. Here, we are given a forward operator $A : \Theta \to \tilde{\Theta}$ and a noisy measurement $f = A\theta + \delta$

where $\delta \in \tilde{\Theta}$ models additive noise. The loss function is then of the form

$$\mathcal{L} = \frac{1}{2} \|A \cdot - f\|_2^2$$

for which one can show that the Bregman iterations converge to a solution of

$$\min \{J(\theta) : A\theta = f\}, \tag{4.22}$$

see, e.g., [Osh+05]. In comparison to this, the concept of adding a regularizing term with parameter $\lambda > 0$, i.e., considering the problem

$$\min_\theta \mathcal{L}(\theta) + \lambda J(\theta)$$

actually modifies the minimizers. In this sense Bregman iterations do not introduce a bias. △

Example 4.21. In order to obtain an intuition of the behavior of Bregman iterations, we consider an image denoising task. I.e., we are given a noisy image $\mathbb{R}^{n\times m} \ni f = u+\delta$ where $\delta \in \mathbb{R}^{n\times m}$ models additive noise. In order to obtain $u \in \mathbb{R}^{n\times n}$ from f we employ the TV functional [ROF92]

$$J(u) = TV(u) := \sum_{i,j} \sqrt{|u_{i+1,j} - u_{i,j}|^2 + |u_{i,j+1} - u_{i,j}|^2}$$

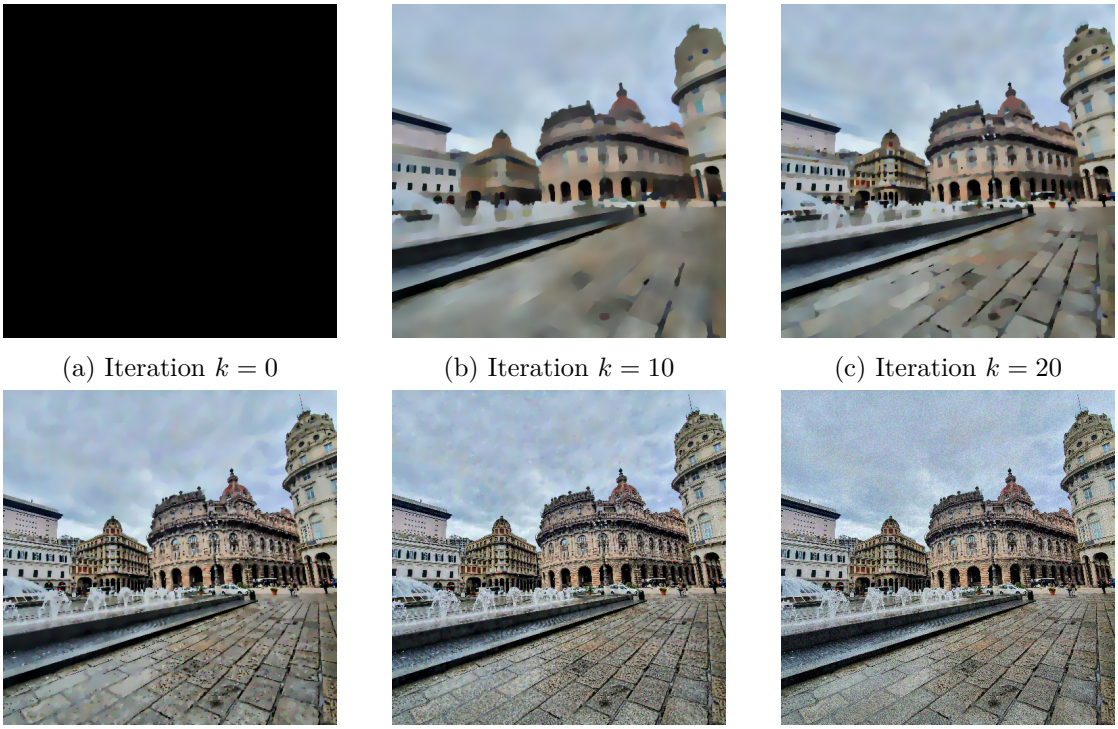
together with the loss function $\mathcal{L}(u) := \frac{1}{2}\|u - f\|_2^2$. We start with an image $u^{(0)}$ such that $TV(u^{(0)}) = 0$, i.e., a constant image. In Fig. 4.3 we visualize the iterates at for different k. At the start, the iterates only display features on a larger scale, while at the end, they converge back to smallest possible scale, the noisy data. In order to obtain an appropriate denoising, one needs to employ a early stopping here. This fits well to the insight from Eq. (4.22) since here the forward operator is the identity, i.e.

$$\left\{u : \frac{1}{2}\|u - f\|^2 = 0\right\} = \{f\}.$$

It should also be noted that this example only serves a explanatory purpose. In practice directly applying Eq. (4.21) for $J = TV$ can become infeasible since the first minimization problem is expensive.

The time continuous flow for $\tau \to 0$ is known as the *inverse scale space* flow [Bur+06; Bur+07],

$$\begin{cases} \dot{p}_t = -\nabla \mathcal{L}(\theta_t), \\ p_t \in \partial J(\theta_t). \end{cases}$$



(a) Iteration $k = 0$ (b) Iteration $k = 10$ (c) Iteration $k = 20$

(d) Iteration $k = 50$ (e) Iteration $k = 100$ (f) Iteration $k = 500$

Figure 4.3.: Bregman iterations for image denoising in Example 4.21.

For $J = \frac{1}{2}\|\cdot\|_2^2$ we see that $\partial J(\theta_t) = \theta_t$ and therefore, we obtain the standard gradient flow. Hence, the inverse scale space flow is a generalization of the standard gradient flow.

4.4.2. Linearized Bregman Iterations and Mirror Descent

The minimization step in Eq. (4.21) is infeasible for large scale applications, especially in our setting for neural networks. Therefore, we employ the idea introduced in [Yin+08; COS09]. We first linearize the loss function around the previous iterate,

$$\mathcal{L}(\theta) \approx \mathcal{L}(\theta^{(k)}) + \left\langle \nabla\mathcal{L}(\theta^{(k)}), \theta - \theta^{(k)} \right\rangle.$$

The next step is to replace J with the strongly convex elastic net regularization

$$J_\delta := J + \frac{1}{2\delta}\|\cdot\|_2^2. \tag{4.23}$$

The minimization step in Eq. (4.21) then transforms to

$$\begin{aligned}
&\operatorname{argmin}_{\theta\in\Theta} D_{J_\delta}^{p^{(k)}}\left(\theta, \theta^{(k)}\right) + \tau\left\langle \nabla\mathcal{L}(\theta^{(k)}), \theta\right\rangle \qquad (4.24)\\
&\quad= \operatorname{argmin}_{\theta\in\Theta} J(\theta) + \frac{1}{2\delta}\|\theta\|_2^2 - \langle p^{(k)}, \theta\rangle + \tau\left\langle \nabla\mathcal{L}(\theta^{(k)}), \theta\right\rangle\\
&\quad= \operatorname{argmin}_{\theta\in\Theta} J(\theta) + \frac{1}{2\delta}\left\|\theta - \delta\left(p^{(k)} - \tau\nabla\mathcal{L}(\theta^{(k)})\right)\right\|_2^2 - \underbrace{\left\|p^{(k)} - \tau\nabla\mathcal{L}(\theta^{(k)})\right\|_2^2}_{\text{constant in } \theta}\\
&\quad= \operatorname{prox}_{\delta J}\left(\delta\left(p^{(k)} - \tau\nabla\mathcal{L}(\theta^{(k)})\right)\right).
\end{aligned}$$

Note, that here $p^{(k)}$ is a subgradient of J_δ at $\theta^{(k)}$, where we derive the subgradient update rule

$$p^{(k+1)} := p^{(k)} - \tau\mathcal{L}(\theta^{(k)}).$$

This yields the linearized Bregman iterations

$$p^{(k+1)} = p^{(k)} - \tau\nabla\mathcal{L}(\theta^{(k)}), \tag{4.25a}$$
$$\theta^{(k+1)} = \operatorname{prox}_{\delta J}(\delta p^{(k+1)}). \tag{4.25b}$$

The last line is equivalent to $p^{(k+1)} \in \partial J_\delta(\theta^{(k+1)})$ for which we obtain the continuous linearized flow

$$\begin{aligned}
\dot{p}_t &= -\nabla\mathcal{L}(\theta_t),\\
p_t &\in \partial J_\delta(\theta_t),
\end{aligned}$$

see [Bur+06; Bur+07].

Connections To Mirror Descent As already noticed in [Vil+23] linearized Bregman iterations are equivalent to mirror descent in some situations. We show the equivalence in the following, where we employ similar arguments as in [BT03]. One assumes to be given a differentiable and strongly convex function $h : \Theta \to \mathbb{R}$, i.e.,

$$h(\overline{\theta}) - h(\theta) - \langle \nabla h(\theta), \overline{\theta} - \theta \rangle \geq \frac{1}{2} \left\| \overline{\theta} - \theta \right\|_2^2$$

for all $\theta, \overline{\theta} \in \Theta$. The mirror descent update then reads ([NY83; BT03])

$$\theta^{(k+1)} = \nabla h^* \left(\nabla h \left(\theta^{(k)} \right) - \tau \mathcal{L}(\theta^{(k)}) \right) \tag{4.26}$$

where h^* denotes the Fenchel conjugate

$$h^*(p) = \sup_{\theta} \langle p, \theta \rangle - h(\theta)$$

with the gradient (see [BV04])

$$\nabla h^*(p) = \operatorname{argmax}_\theta \left\{ \langle p, \theta \rangle - h(\theta) \rangle \right\}.$$

Therefore, we see that Eq. (4.26) can be written as

$$\begin{aligned} \theta^{(k+1)} &= \operatorname{argmax}_\theta \left\{ \left\langle \nabla h \left(\theta^{(k)} \right) - \tau \mathcal{L}(\theta^{(k)}), \theta \right\rangle - h(\theta) \right\} \\ &= \operatorname{argmax}_\theta \left\{ -D_h(\theta, \theta^{(k)}) - \tau \left\langle \mathcal{L}(\theta^{(k)}), \theta \right\rangle \right\} \\ &= \operatorname{argmin}_\theta \left\{ D_h(\theta, \theta^{(k)}) + \tau \left\langle \mathcal{L}(\theta^{(k)}), \theta \right\rangle \right\} \end{aligned}$$

which was our starting point to derive linearized Bregman iterations for $h = J_\delta$ in Eq. (4.24). In fact, we can always find a convex functional $J : \Theta \to \mathbb{R}$ such that $h = J + \frac{1}{2} \|\cdot\|_2^2$. Therefore, we see, that Eq. (4.25) is a more general formulation of Eq. (4.26).

4.4.3. Stochastic and Momentum Variants

We want to employ linearized Bregman iterations to train a neural network. As mentioned in Section 4.1.2, we usually do not compute the full gradient of $\mathcal{L}$ but rather a minibatched variant. This yields stochastic Bregman iterations:

$$\begin{aligned} &\text{draw } \omega^{(k)} \text{ from } \Omega \text{ using the law of } \mathbb{P}, \\ &g^{(k)} := g(\theta^{(k)}; \omega^{(k)}), \\ &v^{(k+1)} := v^{(k)} - \tau^{(k)} g^{(k)}, \\ &\theta^{(k+1)} := \operatorname{prox}_{\delta J}(\delta v^{(k+1)}). \end{aligned} \tag{4.27}$$

We also abbreviate this scheme as the *LinBreg* algorithm in the following. The presence of a stochastic gradient estimator significantly complicates the convergence analysis, as observed in Section 4.4.4. However, this algorithm can now be efficiently employed to train a neural network. For the analogous stochastic mirror descent algorithm we refer to [Nem+09].

Momentum Variant Typically, the learning process of a neural network can be improved by introducing a momentum term (see e.g. [Nes83; Qia99]) in the optimizer. In our case, this can be achieved by replacing the gradient update on the subgradient variable. In [BREG-I] we first consider the inertia version of the gradient flow as

$$\begin{cases} \gamma \ddot{v}_t + \dot{v}_t = -\nabla \mathcal{L}(\theta_t), \\ v_t \in \partial J_\delta(\theta_t). \end{cases}$$

The discretization then reads

$$\begin{aligned} m^{(k+1)} &= \beta^{(k)} m^{(k)} + (1 - \beta^{(k)}) \tau^{(k)} g^{(k)}, \\ v^{(k+1)} &= v^{(k)} - m^{(k+1)}, \\ \theta^{(k+1)} &= \operatorname{prox}_{\delta J}(\delta v^{(k+1)}). \end{aligned} \tag{4.28}$$

Adamized Bregman Iteration We shortly remark that one can replace the momentum update in Eq. (4.28) with an Adam update [KB14]. This then yields an Adamized version of linearized Bregman iterations as employed in [BREG-I].

4.4.4. Convergence of Stochastic Bregman Iterations

While various previous works prove convergence of linearized Bregman iterations (see e.g. [Osh+05; COS09]), the stochastic setting requires special treatment. In [BREG-I] we prove the first guarantees for the algorithm in Eq. (4.27). Other work on convergence of stochastic Bregman iterations, or mirror descent requires a differentiable functional J, see [DEH21; HR21; ZH18; DOr+21; AKL22]. Since our main motivation is a L^1 type functional, this is not applicable. Therefore, we present the novel convergence analysis of [BREG-I].

Assumptions on the Gradient Estimator In order to obtain convergence guarantees, we need to assume mainly two properties on the gradient estimator $g(\cdot, \cdot)$. First we assume unbiasedness, which means

$$\mathbb{E}\left[g(\theta; \omega)\right] = \nabla \mathcal{L}(\theta) \text{ for all } \theta \in \Theta.$$

The second assumption we need in the following is referred to as *bounded variance* of the estimator.

Assumption 4.22 (Bounded variance). We assume that there exists a constant $\sigma > 0$ such that for any $\theta \in \Theta$ it holds

$$\mathbb{E}\left[\|g(\theta; \omega) - \nabla \mathcal{L}(\theta)\|^2\right] \leq \sigma^2. \tag{4.29}$$

Remark 4.23. We remark, that this property is weaker than the bounded gradient assumption

$$\mathbb{E}\left[\|g(\theta;\omega)\|^2\right] \leq C$$

for some constant $C > 0$. In fact this condition can not be enforced together with a strong convexity assumption—which we employ in Theorem 4.30—as shown in [Ngu+18]. △

Assumptions on the Regularizer and on the Loss Function The assumptions on the regularization functional J are mild and merely ensure the well-definedness of the proximal mapping.

Assumption 4.24 (Regularizer). We assume that $J : \Theta \to (-\infty, \infty]$ is a convex, proper, and lower semicontinuous functional on the Hilbert space Θ.

Our assumptions on the loss function $\mathcal{L}$ are more restrictive. We require it to be bounded from below and differentiable, which are both standard assumptions. Additionally, we require Lipschitz continuity of the gradient, which is commonly employed in optimization literature.

Assumption 4.25 (Loss function). We assume the following conditions on the loss function:

- The loss function $\mathcal{L}$ is bounded from below and without loss of generality we assume $\mathcal{L} \geq 0$.
- The function $\mathcal{L}$ is continuously differentiable.
- The gradient of the loss function $\theta \mapsto \nabla\mathcal{L}(\theta)$ is L-Lipschitz for $L \in (0, \infty)$:

$$\left\|\nabla\mathcal{L}(\tilde{\theta}) - \nabla\mathcal{L}(\theta)\right\| \leq L\left\|\tilde{\theta} - \theta\right\|, \quad \forall\theta, \tilde{\theta} \in \Theta. \tag{4.30}$$

If the loss function $\mathcal{L}$ fulfills the previous assumptions we are able to prove loss decay of the iterates, see Theorem 4.29. However, in order to show convergence of the iterates we additionally need a convexity assumption. For a differentiable functional J, the authors in [DEH21] assumed

$$\nu\, D_J(\overline{\theta}, \theta) \leq D_{\mathcal{L}}(\overline{\theta}, \theta),$$

which for twice differentiable J and $\mathcal{L}$ yields

$$\nu\, \nabla^2 J \lesssim \nabla^2 \mathcal{L}, \qquad \forall\overline{\theta}, \theta \in \Theta.$$

Plugging in the definition of the Bregman dist $D_{\mathcal{L}}$ we obtain

$$\nu\, D_J(\overline{\theta}, \theta) \leq \mathcal{L}(\overline{\theta}) - \mathcal{L}(\theta) - \langle\nabla\mathcal{L}(\theta), \overline{\theta} - \theta\rangle.$$

In this form, one observes that this is in fact a convexity assumption on $\mathcal{L}$ in a J dependent distance, as employed in [BREG-I].

Assumption 4.26 (Strong convexity). For a proper convex function $H : \Theta \to \mathbb{R}$ and $\nu \in (0, \infty)$, we say that the loss function $\theta \mapsto \mathcal{L}(\theta)$ is ν-strongly convex w.r.t. H, if

$$\mathcal{L}(\bar{\theta}) \geq \mathcal{L}(\theta) + \langle \nabla \mathcal{L}(\theta), \bar{\theta} - \theta \rangle + \nu D_H^p(\bar{\theta}, \theta), \quad \forall \theta, \bar{\theta} \in \Theta, p \in \partial H(\theta). \tag{4.31}$$

Remark 4.27. We have two relevant cases for the choice of H. For $H = \frac{1}{2} \|\cdot\|^2$ Assumption 4.26 reduces to standard strong ν-convexity. The other relevant case, is $H = J_\delta$, i.e., we consider convexity w.r.t. to the functional J_δ. △

Remark 4.28. In the setting of training a neural network, where we employ the empirical loss Eq. (4.1), this convexity assumption usually fails. While it is possible to enforce this conditions only locally around the minimum, this does not significantly improve the applicability. For future work, it would be desirable to enforce a Kurdyka–Łojasiewicz inequality, as in [Ben+21] for the deterministic case. △

Loss Decay The first convergence result considers the loss decay of the iterates. Here, we do not assume convexity of the loss function. Under this assumptions [Ben+21; BB18] were able to show the inequality

$$\mathbb{E}\left[\mathcal{L}(\theta^{(k+1)})\right] + \frac{1}{\tau^{(k)}} \mathbb{E}\left[D_J^{\mathrm{sym}}(\theta^{(k+1)}, \theta^{(k)})\right] + \frac{C}{2\delta\tau^{(k)}} \mathbb{E}\left[\left\|\theta^{(k+1)} - \theta^{(k)}\right\|^2\right] \leq \mathbb{E}\left[\mathcal{L}(\theta^{(k)})\right].$$

In our setting, employing a stochastic gradient estimator, we are able to prove a similar estimate. We obtain an additional term scaled by σ, which controls the expected squared difference between the gradient estimator and the actual gradient. It should however be noted, that the proof is not only a trivial extension of the one in [BB18].

Theorem 4.29 ([BREG-I, Th. 2]: Loss decay). Assume that Assumptions 4.22, 4.24 and 4.25 hold true, let $\delta > 0$, and let the step sizes satisfy $\tau^{(k)} \leq \frac{2}{\delta L}$. Then there exist constants $c, C > 0$ such that for every $k \in \mathbb{N}$ the iterates of (4.27) satisfy

$$\mathbb{E}\left[\mathcal{L}(\theta^{(k+1)})\right] + \frac{1}{\tau^{(k)}} \mathbb{E}\left[D_J^{\mathrm{sym}}(\theta^{(k+1)}, \theta^{(k)})\right] + \frac{C}{2\delta\tau^{(k)}} \mathbb{E}\left[\left\|\theta^{(k+1)} - \theta^{(k)}\right\|^2\right] \leq \mathbb{E}\left[\mathcal{L}(\theta^{(k)})\right] + \tau^{(k)}\delta\frac{\sigma^2}{2c}. \tag{4.32}$$

Convergence of the Iterates Here, we have two cases respectively proving convergence w.r.t. the L^2 distance and the Bregman distance of J_δ. The first assumes strong convexity with $H = \frac{1}{2} \|\cdot\|^2$ in Assumption 4.26.

Theorem 4.30 ([BREG-I, Th. 6]: Convergence in norm). Assume that Assumptions 4.22, 4.24 and 4.25 and Assumption 4.26 for $H = \frac{1}{2}\|\cdot\|^2$ hold true and let $\delta > 0$. Furthermore, assume that the step sizes $\tau^{(k)}$ are such that for all $k \in \mathbb{N}$:

$$\tau^{(k)} \leq \frac{\mu}{2\delta L^2}, \qquad \tau^{(k+1)} \leq \tau^{(k)}, \qquad \sum_{k=0}^{\infty} (\tau^{(k)})^2 < \infty, \qquad \sum_{k=0}^{\infty} \tau^{(k)} = \infty.$$

The function $\mathcal{L}$ has a unique minimizer θ^* and if $J(\theta^*) < \infty$ the stochastic linearized Bregman iterations (4.27) satisfy the following:

- Letting $d_k := \mathbb{E}\left[D_{J_\delta}^{v^{(k)}}(\theta^*, \theta^{(k)})\right]$ it holds

$$d_{k+1} - d_k + \frac{\mu}{4}\tau^{(k)}\mathbb{E}\left[\left\|\theta^* - \theta^{(k+1)}\right\|^2\right] \leq \frac{\sigma}{2}\left((\tau^{(k)})^2 + \mathbb{E}\left[\left\|\theta^{(k)} - \theta^{(k+1)}\right\|^2\right]\right). \tag{4.33}$$

- The iterates possess a subsequence converging in the L^2-sense of random variables:

$$\lim_{j\to\infty} \mathbb{E}\left[\left\|\theta^* - \theta^{(k_j)}\right\|^2\right] = 0. \tag{4.34}$$

Here, J_δ is defined as in (4.23).

For the second result we assume convexity w.r.t. the Bregman distance, i.e., we choose $H = J_\delta$ in Assumption 4.26. This induces a relation between the Bregman distance of J and the loss function $\mathcal{L}$, which has been similarly employed in [DEH21].

Theorem 4.31 ([BREG-I, Th. 11]: Convergence in the Bregman distance). Assume that Assumptions 4.22, 4.24 and 4.25 and Assumption 4.26 for $H = J_\delta$ hold true and let $\delta > 0$. The function $\mathcal{L}$ has a unique minimizer θ^* and if $J(\theta^*) < \infty$ the stochastic linearized Bregman iterations (4.27) satisfy the following:

- Letting $d_k := \mathbb{E}\left[D_{J_\delta}^{v^{(k)}}(\theta^*, \theta^{(k)})\right]$ it holds

$$d_{k+1} \leq \left[1 - \tau^{(k)}\nu\left(1 - \tau^{(k)}\frac{2\delta^2 L^2}{\nu}\right)\right] d_k + \delta(\tau^{(k)})^2\sigma^2. \tag{4.35}$$

- For any $\varepsilon > 0$ there exists $\tau > 0$ such that if $\tau^{(k)} = \tau$ for all $k \in \mathbb{N}$ then

$$\limsup_{k\to\infty} d_k \leq \varepsilon. \tag{4.36}$$

- If $\tau^{(k)}$ is such that

$$\lim_{k\to\infty} \tau^{(k)} = 0 \quad \text{and} \quad \sum_{k=0}^{\infty} \tau^{(k)} = \infty \tag{4.37}$$

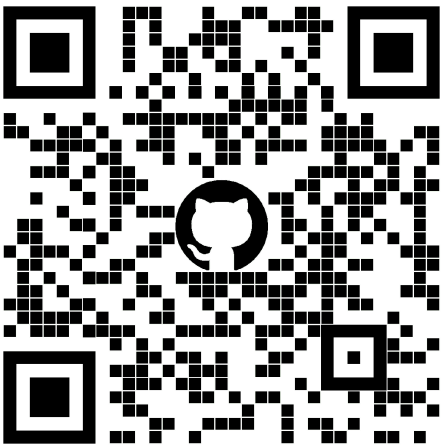
then it holds

$$\lim_{k\to\infty} d_k = 0. \tag{4.38}$$

Here, J_δ is defined as in (4.23).

4.4.5. Numerical Results and Practical Considerations

Before briefly reviewing the numerical results in [BREG-I, Sec. 4], we remark on some practical considerations. In particular we comment on the parameter initialization strategy. All the experiments were implemented in `Python` [VD95] employing—among others—the `PyTorch` package [Pas+19].



The code for all the experiments is available at github.com/TimRoith/BregmanLearning.

Parameter Initialization As already noticed in [GB10] parameter initialization has a significant impact on the training of the neural network. Here, in contrast to standard Bregman methods, we are not able to initialize the parameters of the neural network as $\theta = 0$. This is due to that fact, that a zero initialization induces symmetries in the network weights, for which one cannot utilize the full expressivity of the architecture [GBC16, Ch. 6]. Therefore, we rather employ the approach from [Liu+21; DZ19; Mar10] of sparsifying weight matrices $\tilde{W}^l \in \mathbb{R}^{n_{l+1}\times n_l}$ up to a certain level, by a pointwise multiplication with a binary mask $M^l \in 0, 1^{n_{l+1}\times n_l}$

$$W^l := \tilde{W}^l \odot M^l.$$

Each entry in M^l is i.i.d. sampled from a Bernoulli distribution

$$M^l_{i,j} \sim \mathcal{B}(r).$$

where the parameter r determines the sparsity,

$$\mathrm{N}(W^l) := \frac{\left\|W^l\right\|_0}{n_l \cdot n_{l-1}} = 1 - \mathrm{S}(W^l)$$

with N denoting the percentage of used parameters and S the sparsity. In [GB10] the authors advise to especially control to the variance of the parameter initialization distribution, for which in [BREG-I] we derive

$$\operatorname{Var}\left[\tilde{W}^l\right] = \frac{1}{r} \operatorname{Var}\left[\tilde{W}^l \odot M^l\right] \tag{4.39}$$

and therefore scale the weights with the sparsity parameter r at initialization.

Choice of Regularizers In all our experiments we choose a L^1 type sparsity promoting regularization function functional J. We do not employ any coupling between weight matrices of different layers, and therefore for $\theta = ((W_1, b_1), \ldots, (W_L, b_L))$ we have

$$J(\theta) = \sum_{l=1}^{L} J_l(W_l)$$

where J^l is choosen according to the layer type. In the easiest case of a fully connected layer, we can choose

$$J_l(W_l) := \|W_l\|_1 .$$

In the case of a convolutional layer we have that W^l is determined by convolutional kernels $K_{i,j} \in \mathbb{R}^{k \times k}$, see Section 4.1.1. Here, we typically employ a group sparsity term in the form

$$J_l(W_l) = \|W_l\|_{2,1} = \sum_{i,j} \|K_{i,j}\|_2 .$$

The outer sum acts as a L^1 regularizer on the instances $\|K_{i,j}\|$. Sparsity in this sense, then amounts to having indices (i, j) for which $\|K_{i,j}\|_2 = 0 \Leftrightarrow K_{i,j} = 0$, i.e., we prune away whole convolutional filters. This effect is displayed in [BREG-I, Fig. 1]. We can also employ group sparsity on fully connected layers, by considering row sparsity of $W_l \in \mathbb{R}^{n_{l+1}, n_l}$s

$$J_l(W_l) = \sum_{i=1}^{n_{l+1}} \|W_{i,:}\|_2 = \sum_{i=1}^{n_{l+1}} \sqrt{\sum_{js=1}^{n_l} W_{i,j}^2}.$$

In this setting we have a L^1 penalty on the row norms $\|W_{i,:}\|_2$ which therefore enforces whole rows to be zero. This is relevant, if we employ a layer architecture with $\Psi^l(0) = 0$, e.g., using no bias vectors and the ReLU activation function. In this setting, if the ith row of W_l is zero this effectively means, that the ith neuron in layer $l+1$ is inactive. This observation allows the neural architecture search in one of the following paragraphs.

Comments on the Numerical Results We briefly remark on the numerical results as displayed in [BREG-I, Sec. 4]. In the experiments we employed feed-forward networks with simple linear, convolutional and residual layers and tested on the three datasets [Kri09; XRV17; LC10].

The basic comparison between the algorithms SGD, ProxGD and LinBreg shows the qualitative behavior of each iterations. The sparse initialization does not have any effect on SGD, since it does not preserve the sparsity in any way. ProxGD rather starts with many active parameters and reduces this number during the iteration. Only the discretization of the inverse scale space flow—via Bregman iterations—shows the desired behaviour of gradually adding active parameters. Furthermore, in [BREG-I, Fig. 2] we can see, that the choice of λ in the regularizer $J = \lambda \|\cdot\|_1$ changes the results significantly. In the light of Eq. (4.22) this is not expected for the standard Bregman iterations with a convex loss. It is therefore interesting to see, that in our non-convex and stochastic situation this effect changes.

The momentum variants, as discussed in Section 4.4.3 yield the desired effect of enhancing the validation accuracy, and respectively converging faster. However in each of the experiments, one can also observe that adding a momentum term has the effect that more parameters are added faster. On the one hand this could mean that the network actually requires more parameters to have a higher accuracy, and a momentum variant is more likely to increase the number of needed parameters. However, the quantitative evaluation on the CIFAR10 dataset [Kri09] shows, that especially the Adamized version tends to increase the number of used parameters rather aggressively, while only slightly increasing the performance of the net. The performance here is very similar to the one of proximal gradient descent. However, the training of a residual network seems to be slightly better with a standard Lasso implementation. Neglecting the non-differentiability of the L^1 norm one computes a derivative via automatic differentiation [Ral81; MDA15] (we employed the `autograd` library of the `PyTorch` package [Pas+19]). In order to obtain true zeros in the weight matrix one then has to employ a thresholding operation after the training. In some sense this method is not a proper sparse training approach, but rather a regularization method with an added pruning step at the end.

Comments on Efficiency One of the major advantages of the Bregman approach, is that the network is sparse already during the training time. As with all sparse-to-sparse training approaches this yields a very small number of active parameters over all training step. This sparsity can be easily exploited in each forward pass. However, it is not directly possible to achieve performance gains during the backward pass of the network, since in general

$$W_{ij}^l = 0 \nRightarrow \partial_{W_{ij}^l} \mathcal{L}(\theta) = 0.$$

In [BREG-I; BREG-II] there are no evaluations on the training time and memory consumption of the Bregman algorithm. This is an interesting open question for future work. We remark that the computational complexity of the LinBreg algorithm does not increase significantly, compared to SGD, since the evaluation of the proximal operator is very efficient for L^1 type functionals.