DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2016

The Effect of Batch Normalization on Deep Convolutional Neural Networks

FABIAN SCHILLING

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

KTH Computer Science
and Communication

The Effect of Batch Normalization on Deep Convolutional Neural Networks

FABIAN SCHILLING

Master Thesis at CSC CVAP
Supervisor: Patric Jensfelt
Examiner: Joakim Gustafson

Abstract

Batch normalization is a recently popularized method for accelerating the training of deep feed-forward neural networks. Apart from speed improvements, the technique reportedly enables the use of higher learning rates, less careful parameter initialization, and saturating nonlinearities. The authors note that the precise effect of batch normalization on neural networks remains an area of further study, especially regarding their gradient propagation.

Our work compares the convergence behavior of batch normalized networks with ones that lack such normalization. We train both a small multi-layer perceptron and a deep convolutional neural network on four popular image datasets. By systematically altering critical hyperparameters, we isolate the effects of batch normalization both in general and with respect to these hyperparameters.

Our experiments show that batch normalization indeed has positive effects on many aspects of neural networks but we cannot confirm significant convergence speed improvements, especially when wall time is taken into account. Overall, batch normalized models achieve higher validation and test accuracies on all datasets, which we attribute to its regularizing effect and more stable gradient propagation.

Due to these results, the use of batch normalization is generally advised since it prevents model divergence and may increase convergence speeds through higher learning rates. Regardless of these properties, we still recommend the use of variance-preserving weight initialization, as well as rectifiers over saturating nonlinearities.

Referat

Effekten av batch normalization på djupt faltningsneuronnät

Batch normalization är en metod för att påskynda träning av djupa framåtmatande neuronnnätv som nyligt blivit populär. Förutom hastighetsförbättringar så tillåter metoden enligt uppgift högre träningshastigheter, mindre noggrann parameterinitiering och mättande olinjäriteter. Författarna noterar att den exakta effekten av batch normalization på neuronnät fortfarande är ett område som kräver ytterligare studier, särskilt när det gäller deras gradientfortplantning.

Vårt arbete jämför konvergensbeteende mellan nätverk med och utan batch normalization. Vi tränar både en liten flerlagersperceptron och ett djupt faltningsneuronnät på fyra populära bilddatamängder. Genom att systematiskt ändra kritiska hyperparametrar isolerar vi effekterna från batch normalization både i allmänhet och med avseende på dessa hyperparametrar.

Våra experiment visar att batch normalization har positiva effekter på många aspekter av neuronnät, men vi kan inte bekräfta att det ger betydelsefullt snabbare konvergens, speciellt när väggtiden beaktas. Allmänt så uppnår modeller med batch normalization högre validerings- och testträffsäkerhet på alla datamängder, vilket vi tillskriver till dess reglerande effekt och mer stabil gradientfortplantning.

På grund av dessa resultat är användningen av batch normalization generellt rekommenderat eftersom det förhindrar modelldivergens och kan öka konvergenshastigheter genom högre träningshastigheter. Trots dessa egenskaper rekommenderar vi fortfarande användning av variansbevarande viktinitiering samt likriktare istället för mättande olinjäriteter.

Contents

Contents

Chapter 1

Introduction

Machine learning technologies power many aspects of modern society such as web searches, recommender systems, and content filtering. Their primary focus is to build machines that are not explicitly programmed to perform certain tasks but to learn from labeled data. An example of a supervised machine learning system is a spam filter which decides for an incoming email if it contains spam or not. Rather than explicitly programming an algorithm to look for certain words or phrases, humans can label incoming emails as either spam or non-spam, thus enabling the machine to learn from experience.

Deep learning is a recently popularized approach in the machine learning community that aims to solve tasks such as understanding the content of images, speech signals, and natural language. These types of tasks are very easy for humans but have been notoriously hard for computers since the data is very high dimensional and thus not easily interpretable. The term deep learning can be seen as a rebranding of artificial neural networks with the difference that their architecture consists of more processing layers. The major reason for the abandonment of deeper architectures was the inability to train them effectively. For decades, conventional machine learning techniques were limited in their ability to abstract and process natural data because their architectures were evidently too shallow. Designing such a conventional learning system required careful feature engineering and considerable domain expertise in order to transform the raw data into suitable discriminative representations.

In recent years, various deep learning architectures such as convolutional and recurrent neural networks have overcome their learning difficulties due to structural and algorithmic improvements. The systems have improved the state-of-the-art by a substantial margin in domains such as computer vision [33], speech recognition [22], and natural language processing [7]. One of deep learning's most promising characteristics is the ability to perform automatic feature extraction and model high-level abstractions in various signals such as images, sound, and text.

In the computer vision domain, convolutional neural networks have outperformed traditional methods in tasks such as image classification [33, 4, 20], ob-

ject detection [56, 46, 47], semantic segmentation [42, 19, 13], and face recognition [35, 60]. Instead of handcrafting features and feeding them into shallow statistical models, convolutional neural networks can be seen as a system that performs automatic feature extraction and discrimination end-to-end. Their architecture is inspired by biological processes and loosely based on the neuron responses in the receptive field of the brain's visual cortex [27, 26, 25]. On the one hand, a driving factor for their success have been the abundance of available data via the Internet and huge efforts of the research community to create large hand-labeled datasets such as ImageNet [9]. On the other hand, computer hardware such as GPUs have made giant leaps in terms of throughput and memory capacity such that processing of these large scale datasets becomes feasible.

Ever since the breakthrough results on the ImageNet classification challenge [33] were published, convolutional networks have gained a lot of traction in the research community. Many algorithmic and architectural improvements have been proposed, ranging from the choice of activation function [21, 67, 15], optimization methods [31, 68, 11, 57, 61], regularization techniques [55, 23, 65], to model initialization schemes [37, 14, 21].

A recent improvement called batch normalization [28] matches the state-of-the-art on the ImageNet classification challenge by reducing the model's internal covariate shift and thus significantly accelerates the learning process. By computing batch statistics, it makes normalization an integral part of the model architecture. This change allows for much higher learning rates and thus faster convergence, diminishes the impact of model initialization, and acts as a regularizer. Batch normalization has been adopted by all major deep learning frameworks and continues to improve results in recent research [63, 21]. However, the authors of batch normalization acknowledge that its precise effect on the gradient propagation remains an area of further study and that further theoretical analysis of the algorithm would allow still more improvements and applications [28, p.5, p.8].

In this work, we analyze the effects of the proposed batch normalization procedure on convolutional neural networks. We begin by covering the background knowledge that is necessary to understand the internal workings of such neural networks, followed by an extensive introduction to the theory behind batch normalization. We aim to experimentally analyze the alleged benefits of batch normalization by training different model architectures on popular image datasets. The main objective is to gain an understanding of its effects by comparing the training behavior of batch normalized networks with architectures that lack such normalization.

Chapter 2

Background

The focus of this section is to provide the necessary background information on convolutional neural networks for the image classification task. In case of familiarity with the concepts of convolutional neural networks and image classification, it is advised to continue reading section 3, which aims to motivate batch normalization from a theoretical standpoint.

The goal of image classification is to predict the most likely class label for a given input image among a set of classes. Some common image classification tasks include handwritten digit classification or more general object classification (figure 2.1). A network designed for image classification can easily be extended to solve other tasks such as localization or semantic segmentation, for instance.

(a) Digit classification (b) Object classification

Figure 2.1: Common image classification tasks

At its core, a convolutional neural network is a type of feed-forward neural network that is trained with the backpropagation algorithm. Its parameters are primarily its weights and biases but may also contain other learnable parameters.

In the forward pass, the network computes a nonlinear, differentiable function that is arranged in consecutive layers of different types as in

$$f(\boldsymbol{x}, \boldsymbol{\theta}) = f_\ell(...f_2(f_1(\boldsymbol{x}, \boldsymbol{\theta}_1), \boldsymbol{\theta}_2)...), \boldsymbol{\theta}_\ell) = \hat{\boldsymbol{y}} \tag{2.1}$$

where, for an input $\boldsymbol{x}$, the network computes a set of class scores $\hat{\boldsymbol{y}}$ given its parameters $\boldsymbol{\theta}$ at each of the ℓ layers. For simplicity, we will denote all learnable parameters by $\boldsymbol{\theta}$ without the loss of generality.

Note that each layer may contain a set of hyperparameters which cannot be trained with backpropagation and therefore have to be chosen carefully before the training process. Each layer's output activations act as the input for the following layer. The class scores at the last layer are then compared to the ground truth labels using a loss function. The loss can be seen as a distance metric that quantifies how far away the predictions are from the correct class labels and can be computes as

$$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_i(y_i, \hat{y}_i) \tag{2.2}$$

where $\mathcal{L}$ denotes the overall loss, $\mathcal{L}_i$ the per-example loss, $\hat{\boldsymbol{y}}$ the network's predictions, and $\boldsymbol{y}$ the ground truth labels.

In the backward pass, the network computes the partial derivatives of the loss function with respect to its parameters at each layer. This gradient expression provides the network with information about the direction in which each parameter should be adjusted in order to make the predictions more accurate. The learnable network parameters are therefore updated as in

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L} \tag{2.3}$$

by taking a step of size η in the direction of the negative gradient in order to minimize the expected loss $\mathcal{L}$. This process is called gradient descent.

The property that makes a neural network convolutional is the use of convolutional layers which are the core of its design principle. A typical architecture consists of one or many convolutional layers followed by one or many fully connected layers and a final softmax layer that computes the model's predictions. The softmax output can be interpreted as normalized probabilities assigned to each class label. Most layers are followed by an activation function, which introduces nonlinearity into the network. Nonlinear activation functions are crucial for the network to learn interesting, discriminative representations of the data that maximize the classification accuracy. In order to backpropagate the errors made by the classifier during training, we can feed the network's predictions into a surrogate loss function that assesses the current performance. In practice, a variety of layers and other concepts are utilized, the most important of which are organized as follows.

2.1 Overview

A modern convolutional neural network architecture for image classification is comprised of many different algorithmic and structural components that are essential for its ability to learn effectively and generalize well to unseen data. The following provides a quick overview of all necessary components that are related either directly to batch normalization or are critical for successful network training in general. The choice of any network component discussed below may have consequences throughout the entire network and they have to be selected carefully in order to affect the network's performance in a favorable way.

Section 2.2 defines the type of dataset used for image classification tasks and motivates the use of a training, test, and validation set. The training set is used exclusively to fit the parameters of the network whereas the test set is used to assess the classifier's performance on unseen data at the end of training. The main purpose of the validation set is to provide an unbiased estimate of the classifier's performance during training.

After a suitable dataset has been selected, it is commonly preprocessed as discussed in section 2.3. Data preprocessing has various advantageous effects on both the convergence behavior and generalization performance of neural networks and statistical models in general.

Before the model can start the learning process, its adjustable parameters have to be initialized to sensible values that maximize the effect of the parameter updates. The most common initialization schemes are highlighted in section 2.4.

Section 2.5 illustrates the essential steps that the backpropagation algorithm takes in order to adjust the network parameters over the course of training. The section highlights the intuition behind iterative backpropagation and breaks the procedure into forward pass, loss calculation, backward pass, and the subsequent parameter update with gradient descent.

Section 2.6 provides an introduction to the the most important variants of the gradient descent algorithm, namely batch gradient descent, stochastic gradient descent, and mini-batch stochastic gradient descent. The key take-away is that these variants only differ in the amount of data that is used for a single parameter update.

Section 2.7 discusses several gradient descent update rules and motivates the most common techniques for first-order parameter optimization that are a key part of the backpropagation algorithm. Second-order methods are motivated with regard to their superior properties but discussed only briefly since they are too expensive to compute in an online setting.

Section 2.8 provides an overview of the most important global hyperparameters of neural networks and their selection. Namely, those are the learning rate, the learning rate decay, and the batch size used for mini-batch stochastic gradient descent. Layer specific hyperparameters are discussed in their respective sections.

Section 2.9 discusses the most common layers that are present in a modern convolutional neural network architecture, namely the fully connected, the convolutional, and the pooling layer. It has to be noted that in a modern architecture, the convolutional layers precede the fully connected layers. However, it is crucial to understand fully connected layers and their limitations in order to motivate the existence of convolutional layers.

Activation functions, as discussed in section 2.10, introduce nonlinearity into a neural network. Nonlinear activation functions are crucial for the network's ability to learn complex predictive relationships. We show the problems that arise when using historical saturating activation functions and thus motivate the use of rectified nonlinearities, which are at the core of state-of-the-art convolutional neural networks.

Section 2.11 gives an introduction to the most common objective functions

and motivates them from an intuitive and theoretical standpoint. Historically, the squared error and hinge loss have been the most popular loss functions but have since been replaced by the cross entropy loss due to its empirical performance, as well as its plausible information theoretical and probabilistic interpretation.

Regularization, as discussed in section 2.12, is an important concept to avoid overfitting. We discuss both general approaches such as weight regularization and techniques specifically tailored to neural networks such as dropout.

2.1.1 Notation

As in most scientific disciplines, the notations used when describing concepts can vary significantly from author to author. This section acknowledges these differences and aims to provide a consistent notation for this work.

Scalars, vectors, matrices, and tensors

We denote scalars with x, vectors with bold emphasis as in $\boldsymbol{x}$, and matrices with uppercase letters as in X. There is no consistent notation for general tensors, i.e. matrices with more than two dimensions, and we generally also denote them with the matrix notation.

The most common symbols used to describe artificial neural networks mathematically are $\boldsymbol{w}$ for its weights, $\boldsymbol{b}$ for its biases, and $\boldsymbol{\theta} = (\boldsymbol{w}, \boldsymbol{b}, \cdot)$ for the generalization of both which can include other learnable parameters. Since these parameters are used throughout our work, we will make use of the general vector notation, although the weights in a fully connected layer are two dimensional, or even three dimensional in the case of a convolutional layer.

Gradients and derivatives

The gradient or Jacobian matrix is a generalization of the concept of the derivative of a function in one dimension to a function in several dimensions. In neural networks, we are dealing with high dimensional data, and are interested in the first-order partial derivatives of a function with respect to its parameters as in

$$J = \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}) = \frac{\partial f(\boldsymbol{x}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \begin{pmatrix} \frac{\partial f(x_1, \boldsymbol{\theta})}{\partial \theta_1} & \cdots & \frac{\partial f(x_1, \boldsymbol{\theta})}{\partial \theta_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(x_m, \boldsymbol{\theta})}{\partial \theta_1} & \cdots & \frac{\partial f(x_m, \boldsymbol{\theta})}{\partial \theta_n} \end{pmatrix} \tag{2.4}$$

where $f(\boldsymbol{x}, \boldsymbol{\theta})$ is a vector valued function parametrized by $\boldsymbol{\theta}$. J denotes the $m \times n$ Jacobian matrix of first-order partial derivatives, where m is the number of inputs and n the number of learnable parameters.

For notational brevity we will exclusively make use of the ∇ notation. If it is contextually clear with respect to which variables the partial derivatives are taken, we will use the shorthand notation ∇f and omit $\boldsymbol{x}$ and $\boldsymbol{\theta}$ for clarity.

2.2 Datasets

Choosing the right dataset for a given classification task is of crucial importance for the generalization performance. Ultimately, the model's ability to classify unseen data correctly is highly dependent on the quality of the underlying dataset itself. For an image classification problem, the network's input X consists of a variety of grayscale or color images and their corresponding ground truth labels $\boldsymbol{y}$. An image X_i can be viewed as a $c \times h \times w$ tensor of numbers with c color channels and corresponding height h and width w. For a color image, its three channels correspond to the red, green, and blue (RGB) intensities at each pixel. Conversely, for a grayscale image, its single channel corresponds to the grayscale intensity.

For any supervised machine learning task, it is common to split the entire dataset into three subsets called the training set, the test set, and the validation set. Figure 6.1 provides a schematic overview of these dataset splits. They each serve a different purpose and are motivated as follows.

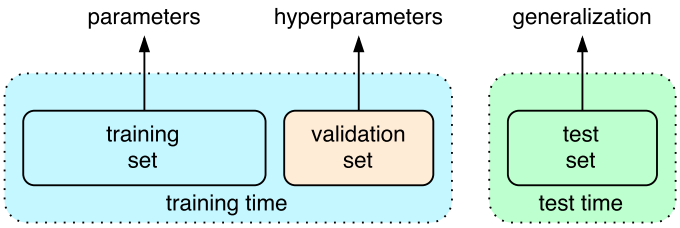


Figure 2.2: Schematic overview of training, validation, and test set.

2.2.1 Training set

The training set is typically the largest of the three and is used to find the model parameters that best explain the underlying predictive relationship between the data and its labels. Most approaches that fit parameters based on empirical relationships with the training set alone tend to overfit the data, meaning that the can identify apparent relationships in the training data but fail to do so in general. This motivates the use of a test set.

2.2.2 Test set

The test set is a set of data that is not used during training, but follows the same probability distribution and predictive relationship. If a model performs well on the training set and also fits the test set well, i.e. predicts the correct label for a large number of unseen input data, minimal overfitting has taken place. It is important to note that the test set is usually only employed once as soon as the model's parameters and hyperparameters are fully specified in order to assess the model's generalization performance. However, to approximate a model's predictive performance during training, a validation set is used.

2.2.3 Validation set

The validation is created by splitting the training set in two parts, the smaller of which is commonly used for hyperparameter optimization and to avoid overfitting through early stopping.

Hyperparameters are discussed further in section 2.8 and refer to the model parameters that cannot be adjusted with backpropagation. In order to choose favorable hyperparameters and maximize the model's predictive performance, a validation set is used as an additional dataset for two reasons. If the training set were used to fit the hyperparameters and parameters simultaneously, the model tends to overfit to the training set. On the contrary, if the hyperparameters were chosen with respect to the test set, the model implicitly fits the assumptions of the test data which would mitigate the unbiased assessment of its predictive performance and therefore its generality. These limitations sufficiently motivate the use of of a third set for validation.

Early stopping is a universally applicable attempt to find the optimal time to stop the training process and to decide when a model is fully specified with respect to its parameters and hyperparameters. A common technique is to continuously monitor the so-called validation accuracy and to stop training when it stops improving. The validation accuracy can be seen as an approximation of the final classification performance that imposes minimal assumptions on the data's predictive relationship, both theoretically and in practice.

2.3 Data preprocessing

Data preprocessing is an important step in any machine learning problem that aims to transform the raw input features into a feature space that is easily interpretable by a machine. The most common preprocessing steps are standardization and whitening [38]. In the following sections, X denotes the dataset, μ the population mean, and σ the population standard deviation.

2.3.1 Standardization

Standardization is the most popular form of preprocessing that is commonly comprised of mean subtraction and subsequent scaling by the standard deviation. The reason for mean subtraction is mainly that non-zero mean input data creates a loss surface that is steep in some directions and shallow in other such that it slows down convergence of gradient-based optimization techniques. Conversely, input data that has a large variation in spread along different directions negatively affects the convergence rate [38].

Mean subtraction can be formalized as

$$\begin{aligned} \mu &= \frac{1}{n}\sum_{i=1}^{n} X_i \\ X^{(c)} &= X - \mu \end{aligned} \tag{2.5}$$

where μ denotes the mean and $X^{(c)}$ the zero centered data.

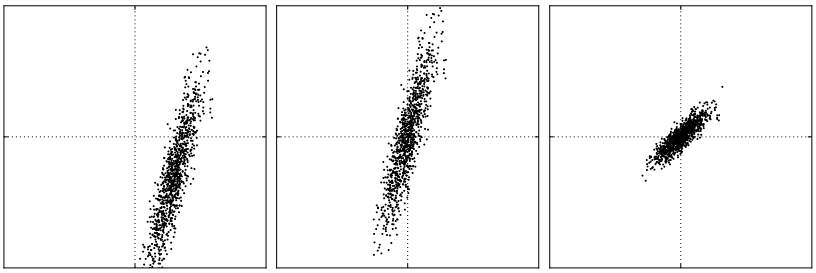
Mean subtraction has the geometric interpretation of centering the cloud of data around the origin along every dimension (figure 2.3b).

Standardization refers to altering the data dimensions such that they are of approximately the same scale. This is commonly achieved by dividing each dimension by its standard deviation once it has been zero centered as in

$$\begin{aligned} \sigma^2 &= \frac{1}{n}\sum_{i=1}^{n}(X_i - \mu)^2 \\ X^{(s)} &= \frac{X - \mu}{\sigma} \end{aligned} \tag{2.6}$$

where σ denotes the standard deviation and $X^{(s)}$ the standardized data.

Dividing by the standard deviation has the geometric interpretation of altering the spread of the data such that the data dimensions are proportional to each other (figure 2.3c).



(a) Original data (b) Zero centered data (c) Standardized data

Figure 2.3: Visualization of the standardization transform.

2.3.2 Whitening

It is sometimes not enough to center and scale the features independently using the standardization process, since a downstream model can further make assumptions on the linear independence of the features. To address this issue, we can make use of the whitening transformation to further remove the linear correlation across features. There are many possible ways to obtain a whitening transformation such as zero component analysis (ZCA) or principal component analysis (PCA). We will

focus on the latter, since PCA is also widely used in the machine learning for the purpose of dimensionality reduction. The whitening transformation is a two-step process that involves decorrelation using the computed eigenvectors, and subsequent scaling of the decorrelated data with the eigenvalues.

The first step of PCA whitening is to perform the singular value decomposition (SVD) of the covariance matrix as in

$$\begin{aligned} \Sigma &= \frac{1}{n} X X^T \\ U, S, U^T &= \text{SVD}(\Sigma) \end{aligned} \tag{2.7}$$

where Σ denotes the covariance matrix, U its eigenvectors, and S contains the corresponding eigenvalues along its diagonal.

In order to decorrelate the data, we compute the dot product of the zero centered data with its eigenvectors as in

$$X^{(d)} = XU \tag{2.8}$$

where U is the matrix of eigenvectors and $X^{(d)}$ denotes the decorrelated data.

Correlated input data usually causes the eigenvectors to be rotated away from the coordinate axes and thus weight updates are not decoupled. However, adaptive optimization method (section 2.7.2) mitigate the need for decorrelation since the weights are updated at different rates independently. The geometric interpretation of decorrelation is to align the data in the directions of the most variance (figure 2.4b).

The whitening operation takes the data in the eigenbasis and divides every dimension by its eigenvalue to normalize the scale as in

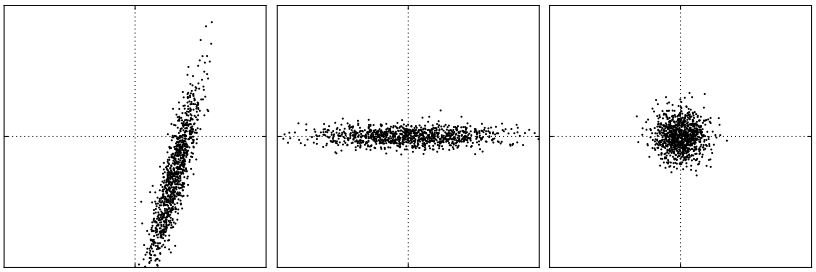
$$X^{(w)} = \frac{XU}{\sqrt{S + \epsilon}} \tag{2.9}$$

where S is the matrix of eigenvalues and $X^{(w)}$ denotes the whitened data. The factor ϵ is a small number that is added to avoid division by zero and therefore provides numerical stability.

The geometric interpretation of this transformation is that if the input data is a multivariate gaussian, then the whitened data will be gaussian with zero mean and identity covariance matrix (figure 2.4c).

2.4 Model initialization

Before the neural network training can begin, its parameters have to be initialized to sensible values. For the biases, a common initialization method is all zeros. For the weights, however, the all zero initialization would cause every neuron in the network to compute the same output, therefore computing the exact same gradients during backpropagation. Symmetry breaking weight initialization is very important for a neural network to train effectively. In the following sections, the most common



(a) Original data (b) Decorrelated data (c) Whitened data

Figure 2.4: Visualization of the whitening transform.

weight initialization schemes are discussed in detail. We generalize to the normal distribution but a uniform distribution is equally suitable when scaled accordingly.

Some authors initialize the biases to small positive values to mitigate the adverse effect of dead neurons when using rectified activation function. The positive bias values push the activations towards the linear regime of the nonlinearity and thus avoid killing activations caused by random gradient fluctuations in the beginning of training.

2.4.1 LeCun initialization

A historical initialization scheme - called LeCun initialization [37] in most deep learning frameworks - proposes to sample weights from a multinomial normal distribution with zero mean and a small standard deviation to break the symmetry that arises from all zero initialization. LeCun initialization can be formalized as

$$\boldsymbol{w}_0 = \mathcal{N}\left(\mu = 0, \sigma^2 = \alpha\right) \tag{2.10}$$

where α can be any positive number.

The gradients computed during backpropagation are proportional to its weights, i.e. a layer with small initial weights will compute small gradients during backpropagation, whereas larger weights lead to larger gradients.

2.4.2 Xavier initialization

The problem with LeCun initialization is that the distribution of outputs from a randomly initialized neuron has a variance that grows with the number of inputs. A normalization scheme for saturating nonlinearities (section 2.10.1) called Xavier initialization [14] scales the weights by the average of the number of its inputs and outputs, achieving the desired output variance of one during forward the forward and backward pass. The Xavier initialization scheme can be formalized as

$$\boldsymbol{w}_0 = \mathcal{N}\left(\mu = 0, \sigma^2 = 2/(n_{\text{in}} + n_{\text{out}})\right) \tag{2.11}$$

where n_{in} and n_{out} denote the number of inputs and outputs of the current layer, respectively.

2.4.3 Kaiming initialization

An initialization method specifically tailored to networks using rectified nonlinearities is called Kaiming initialization [21]. In particular, a rectified linear unit (section 2.10.2) is zero for half of its input, therefore the weight variance must be doubled to keep the variance of the activations constant. This method is the de-facto standard of weight initialization when using rectified nonlinearities and can be formulated as

$$\boldsymbol{w}_0 = \mathcal{N}\left(\mu = 0, \sigma^2 = 2/n_{\text{in}}\right) \tag{2.12}$$

where n_{in} denotes the number of inputs of the current layer.

2.5 Backpropagation

Backpropagation – or backward propagation of errors – is the central algorithm used for training neural networks and is used in conjunction with gradient descent optimization methods (section 2.7).

During the forward pass, the algorithm computes the network's predictions based on the current parameters. Consequently, the predictions are fed into the loss function (section 2.11) and compared to the corresponding ground truth labels.

During the backward pass, the model computes the gradient of the loss function with respect to the current parameters, after which the parameters are updated by taking a step of size η in the direction of minimized loss. The essential steps of the backpropagation procedure are highlighted in algorithm 1.

The `forward` and `backward` functions in algorithm 1 are shorthands for calling the respective function for each of the network's layers. In the `forward` pass, the execution starts by feeding the inputs through the first layer, thus creating the output activations for the subsequent layer. This process is repeated until the loss function at the last layer is reached. During the `backward` pass, the last layer computes the gradients with respect to its own learnable parameters (if any) and also with respect to its own input, which serves as the upstream derivatives for the previous layer. This process is repeated until the input layer is reached.

2.6 Gradient descent

Once the gradients of the loss with respect to the parameters are computed with backpropagation, they are used to perform a gradient descent parameter update. There are three popular variants of gradient descent, which differ in how much data

Algorithm 1: Backpropagation with stochastic gradient descent (SGD)

Data: input $\boldsymbol{x}$, labels $\boldsymbol{y}$, parameters $\boldsymbol{\theta}$, learning rate η
Result: trained model

$e \leftarrow 1$ `# current epoch`
repeat
  $l \leftarrow 0$ `# total loss`
  for $\boldsymbol{x}_i, \boldsymbol{y}_i \in \{\boldsymbol{x}, \boldsymbol{y}\}_{i=1}^{n}$ **do**
    $\hat{\boldsymbol{y}}_i \leftarrow \texttt{forward}(\boldsymbol{x}_i, \boldsymbol{\theta})$ `# forward pass`
    $l_i \leftarrow \texttt{loss}(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i)$ `# loss calculation`
    $l \leftarrow l + l_i$ `# update total loss`
    $\nabla_{\boldsymbol{\theta}} \mathcal{L}_i \leftarrow \texttt{backward}(\boldsymbol{x}_i, \hat{\boldsymbol{y}}_i, \boldsymbol{\theta})$ `# backward pass`
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}_i$ `# gradient descent`
  end
  $l \leftarrow l/n$ `# estimate epoch loss`
  $e \leftarrow e + 1$
until *convergence*

we use to compute the gradient of the loss function with respect to the network parameters. Depending on the amount of data, we make a tradeoff between the accuracy of the parameter update and the time it takes to perform an update.

The simplest form of parameter update is to change the parameters along the negative gradient direction. Since the gradient indicates the direction of increase, it is negated such that the loss function is minimized and not maximized. Mathematically all three gradient descent updates have the following form

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L} \tag{2.13}$$

where η denotes the learning rate, a hyperparameter that controls the step size of a single update.

2.6.1 Batch gradient descent

Batch gradient descent computes the gradient of the loss function for the entire training dataset $\boldsymbol{x}$ and labels $\boldsymbol{y}$. Since the gradients have to be calculated for the whole dataset to perform a single parameter update, batch gradient descent can be very slow and is intractable for datasets that do not fit in memory. Batch gradient descent also does not allow the model to be updated online, i.e. with new examples on the fly.

2.6.2 Stochastic gradient descent

In contrast, stochastic gradient descent (SGD) performs a parameter update for each training example $\boldsymbol{x}_i$ and label $\boldsymbol{y}_i$. SGD therefore performs redundant com-

putations for large datasets, as it recomputes gradients for similar examples before each parameter update. Its frequent updates have a high variance and can cause the loss function to fluctuate heavily. The convergence of stochastic gradient descent has been analyzed using the theories of convex optimization and stochastic approximation and has a high chance of reaching a global or local minimum.

2.6.3 Mini-batch stochastic gradient descent

Mini-batch stochastic gradient descent finally takes the best of both worlds and performs and update for every mini-batch of training examples $\boldsymbol{x}_{\mathcal{B}}$ and its corresponding labels $\boldsymbol{y}_{\mathcal{B}}$. The hyperparameter $\mathcal{B}$ denotes the batch size, i.e. how many training examples are processed simultaneously to perform a single parameter update. Mini-batch SGD reduces the variance of the parameter updates, which can lead to more stable convergence. Secondly, we can make use of highly optimized and parallelized matrix operations which are frequently used in modern deep learning frameworks. Mini-batch gradient descent is the de-facto standard algorithm for training deep neural networks and the term SGD is usually employed also when mini-batches are used, which can sometimes lead to confusion.

2.7 Optimization methods

Several improvements to the standard mini-batch stochastic gradient descent update rule exist and are in frequent use today. The most important of these improved optimization techniques are explained in the following sections. Although second-order methods have stronger convergence properties, we will focus on first-order global and adaptive optimization methods since they are feasible to compute in an online setting.

2.7.1 Momentum

Momentum update [49] is an optimization approach that results in faster convergence rates of stochastic gradient descent in deep networks. The momentum update rule can be motivated from a physical perspective of the optimization problem. In particular, the loss can be interpreted as the height of a hilly terrain and the optimization process can be seen as equivalent to simulating a ball – i.e. the parameter values – rolling downwards on a landscape. The difference to an SGD update is that the gradient only directly influences the velocity, which in turn has an effect on the position. The momentum update rule can be expressed as

$$\begin{aligned} \boldsymbol{v} &\leftarrow \mu \boldsymbol{v} - \eta \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L} \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \boldsymbol{v} \end{aligned} \tag{2.14}$$

where $\boldsymbol{v}$ is the velocity that accumulates the gradients over time and the hyperparameter μ is referred to as momentum and consistent with the physical meaning of friction.

Nesterov momentum

Nesterov momentum [57] is another popular update rule that is inspired by Nesterov's accelerated gradient and proposes a slightly different version of the regular momentum update. It enjoys stronger theoretical convergence guarantees for convex functions and performs better than regular momentum update in practice. The core idea is that when the current parameter vector $\boldsymbol{\theta}$ is at some position, we know that the momentum term alone is slightly affecting the parameter vector by $\mu\boldsymbol{v}$. Therefore, we can treat the future approximation position $\boldsymbol{\theta} + \mu\boldsymbol{v}$ as a look ahead as in

$$\begin{aligned} \boldsymbol{v} &\leftarrow \mu\boldsymbol{v} - \eta \cdot \nabla_{(\boldsymbol{\theta}+\mu\boldsymbol{v})}\mathcal{L} \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \boldsymbol{v} \end{aligned} \tag{2.15}$$

where all variables correspond to the ones of the regular momentum update, except that the gradient is evaluated at $\boldsymbol{\theta} + \mu\boldsymbol{v}$ instead of the old position $\boldsymbol{\theta}$.

A visual comparison of regular momentum and Nesterov momentum can be obtained in figure 2.5.

(a) Momentum update (b) Nesterov momentum update

Figure 2.5: Momentum vs. Nesterov momentum update rule

2.7.2 Adaptive methods

The momentum improvements of SGD discussed so far manipulate all parameters equally since they employ a global learning rate. In this section, some commonly used adaptive optimization methods are highlighted. The cache variables $\boldsymbol{c}$ in the adaptive methods below are to be interpreted as a tensor of the same shape as the parameters $\boldsymbol{\theta}$ and are utilized to perform per-parameter updates.

Adagrad

Adagrad [11] is a subgradient method that dynamically incorporates knowledge of the geometry of the data in form of a cache variable $\boldsymbol{c}$. This cache can be used to perform more informative gradient-based learning based on the updates in earlier iterations as in

$$\begin{aligned} \boldsymbol{c} &\leftarrow \boldsymbol{c} + (\nabla_{\boldsymbol{\theta}}\mathcal{L})^2 \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \eta \cdot \frac{\nabla_{\boldsymbol{\theta}}\mathcal{L}}{\sqrt{\boldsymbol{c} + \epsilon}} \end{aligned} \tag{2.16}$$

where the cache variable $\boldsymbol{c}$ has the same shape as the gradient and keeps track of the per-parameter sum of squared gradients, thus normalizing the parameter update step element-wise. The smoothing term ϵ is a small number that is used to avoid division by zero, and thus provides numerical stability.

Adadelta

The Adadelta [68] and RMSProp [61] parameter updates adjust the Adagrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate. Both algorithms have been developed independently around the same time stemming from the need to resolve Adagrad's diminishing learning rates. RMSProp is based on the RProp [48] algorithm and extends it to be used with mini-batch stochastic gradient descent. In particular, it uses a moving average of squared gradients, which can be formulated as

$$\begin{aligned} \boldsymbol{c} &\leftarrow \lambda\boldsymbol{c} + (1-\lambda)(\nabla_{\boldsymbol{\theta}}\mathcal{L})^2 \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \eta\frac{\nabla_{\boldsymbol{\theta}}\mathcal{L}}{\sqrt{\boldsymbol{c} + \epsilon}} \end{aligned} \tag{2.17}$$

where λ is a hyperparameter that denotes the decay rate.

The update is identical to Adagrad but the cache is leaky, meaning that Adadelta and RMSProp still modulate the learning rate of each parameter based on the magnitude of its gradients. This still has the beneficial equalizing effect but unlike Adagrad, the updates do not become monotonically smaller.

Adam

Adam [31] is a recently published adaptive optimization technique that has been widely adopted. It combines the benefits of RMSProp and Adadelta with those of the momentum update by incorporating a moving average of both the gradient magnitude and its direction as in

$$\begin{aligned} \boldsymbol{c}_1 &\leftarrow \alpha\boldsymbol{c}_1 + (1-\alpha)(\nabla_{\boldsymbol{\theta}}\mathcal{L}) \\ \boldsymbol{c}_2 &\leftarrow \beta\boldsymbol{c}_2 + (1-\beta)(\nabla_{\boldsymbol{\theta}}\mathcal{L})^2 \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \eta \cdot \frac{\boldsymbol{c}_1}{\sqrt{\boldsymbol{c}_2 + \epsilon}} \end{aligned} \tag{2.18}$$

where α and β are hyperparameters control the speed of the decay.

The full Adam update includes a bias correction mechanism, which compensates for the fact that in the first time steps $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ are both initialized and therefore biased at zero. The update is performed in the same way as Adadelta or RMSProp,

except the smooth version of the gradient is used instead of the raw gradient vector. A slightly different variant of Adam is called Adamax and is based on the infinity norm L_p instead of the L_2 norm.

2.7.3 Second-order methods

Second-order methods speed up the training by estimating not only the gradient but also the curvature of the loss surface. Given the curvature, one can estimate the approximate location of the actual minimum, as opposed to taking an uninformed step into the direction of the negative gradient. So far, we have only incorporated the so-called Jacobian matrix J into the update steps, i.e. the matrix containing all the first partial derivatives with respect to the parameters

$$J_{ij} = \nabla_{\boldsymbol{\theta}} \mathcal{L} = \frac{\partial \mathcal{L}_i}{\partial \theta_j} \tag{2.19}$$

where J is a $n \times m$ matrix of first-order derivatives with n number of inputs and m parameters.

Ideally, we would like to take the information given by the Hessian matrix H, i.e. the matrix containing all the second partial derivatives with respect to the parameters, into account

$$H_{ij} = \nabla^2_{\boldsymbol{\theta}} \mathcal{L} = \frac{\partial^2 \mathcal{L}}{\partial \theta_i \partial \theta_j} \tag{2.20}$$

where H is a $n \times n$ matrix of second-order derivatives and n is the number of parameters.

The parameter update then becomes an expression that incorporates knowledge about the loss curvature, which is essentially Newton's method for iterative optimization

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \cdot H^{-1} \nabla_{\boldsymbol{\theta}} \mathcal{L} \tag{2.21}$$

where H^{-1} denotes the inverse of the Hessian matrix.

The core problem for incorporating second-order methods into deep neural networks is that the Hessian is a $n \times n$ matrix, where n denotes the number of learnable parameters. Computing the inverse of such a large matrix is a very costly procedure and second-order methods are therefore not feasible when the number of learnable parameters is high. Another limitation of second-order methods is that they are often only appropriate if the loss function is convex, which is usually not the case in a neural network setting.

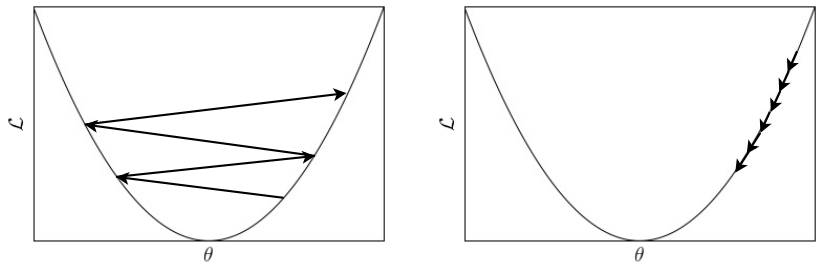
In order to avoid the costly computation of the inverse Hessian matrix, several approximation algorithms have been developed. Most notably, the Conjugate Gradients (CG) method and the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm. However, a rigorous investigation of these methods is beyond the scope of this thesis and shall remain as additional information to the reader.

2.8 Hyperparameters

Hyperparameters and their selection are very important concepts in machine learning, especially in the context of neural networks since these types of models employ a variety of them. Intuitively, hyperparameters can be seen as knobs that have to be tuned independently of the model parameters, and are typically chosen before the learning process begins. The following sections provide an overview of the most important global hyperparameters of feed-forward neural networks trained with backpropagation and stochastic gradient descent. Optimization-specific hyperparameters, layer-specific hyperparameters, and regularization-specific hyperparameters are discussed in sections 2.7, 2.9, 2.12, respectively.

2.8.1 Initial learning rate

The initial learning rate η_0 is arguably the most important hyperparameter when training neural networks. On the one hand, if the learning rate is too large, the average loss will either increase and cause the optimization procedure to diverge or simply miss favorable local minima (figure 2.6a). On the other hand, if the learning rate is too low, the network might take much longer to train to reach the same minimum it could have reached with a higher learning rate in less time (figure 2.6b). A good heuristic for finding the optimal learning rate is to choose the largest learning rate that does not cause divergence of the training process [2]. A schematic intuition of suboptimal learning rates can be obtained in figure 2.6.



(a) Learning rate too high (b) Learning rate too low

Figure 2.6: The effect of suboptimal learning rates on the loss

2.8.2 Learning rate decay

In training deep networks, learning rate decay can be seen as a form of regularization, and it is usually helpful to avoid overfitting. With a high learning rate, the system contains too much kinetic energy and the parameter vector behaves chaotically such that favorable minima are missed. There are many types of learning rate decay, the most important of which are discussed in the sections below. It is important to note that there is no single best solution for annealing the learning rate,

and the choice is ultimately very dependent on the underlying model architecture and the dataset.

We define an iteration as a single pass over one mini-batch of data and an epoch as a pass over all mini-batches, i.e. the entire training set.

Linear decay

Linear decay decreases the learning rate linearly over time until a minimum learning rate is reached after a fixed number of iterations or epochs as in

$$\eta_{t+1} = \eta_0 - (t \cdot (\eta_0 - \eta_{\min})/t_{\text{sat}}) \tag{2.22}$$

where η_0 is the initial learning rate, $\eta_{\min}$ is the minimum learning rate, and t_{sat} is the number of epochs or iterations at which the minimum learning rate should be reached.

Exponential decay

Exponential decay decreases the learning rate proportional to its current value and is closely related to the notion of mean lifetime and half-life as in

$$\eta_{t+1} = \eta_0 \cdot e^{-kt} \tag{2.23}$$

where η_0 is the initial learning rate, k is a hyperparameter that controls the speed of the decay, and t is the number of iterations or epochs.

Power decay

Power decay – sometimes called $1/t$ decay – has a similar formulation to exponential decay but anneals the learning rate less aggressively as in

$$\eta_{t+1} = \eta_0 \cdot (1 + kt)^{-1} \tag{2.24}$$

where η_0 is the initial learning rate, k the decay speed, and t the number of iterations or epochs.

Adaptive decay

Adaptive decay – or sometimes called step decay – scales the learning rate by a constant factor every few epochs as in

$$\eta_{t+1} = k \cdot \eta_t \tag{2.25}$$

where k is the decay factor.

A heuristic used in practice is to watch the validation loss while training with a fixed learning rate and reduce the learning rate by a factor k whenever the validation error stops improving.

Scheduled decay

Another popular method for annealing the learning rate is to specify a manual schedule, either for specific epochs or iterations. This technique is often employed once the dynamics of convergence for a specific model and dataset have been explored thoroughly.

2.8.3 Batch size

In theory, the batch size should not affect the convergence behavior of mini-batch stochastic gradient descent in significant ways. Larger batch sizes should skew the convergence behavior towards batch gradient descent, whereas smaller batch sizes should result in a behavior similar to pure stochastic gradient descent. The impact of the batch size is mostly of computational nature since modern hardware can parallelize matrix multiplications and convolutions, for instance.

However, it has to be noted that the batch size directly affects the batch normalization process since the statistics may vary significantly when choosing smaller batch sizes. The effect of the batch size hyperparameter shall therefore be examined with respect to batch normalization. We refer the reader to section 3 for the theoretical background on batch normalization.

2.8.4 Hyperparameter selection

Hyperparameter selection is usually performed with one of the following techniques: manual search, grid search, and random search.

Manual search essentially refers to picking one hyperparameters intuitively and testing its performance on the validation set.

Grid search refers to a technique were multiple hyperparameters such as the learning rate and the batch size are picked within fixed intervals and step sizes in order to maximize performance on the validation set.

Random search refers to hyperparameter selection with fixed intervals but random step sizes and has several favorable properties as opposed to manual or grid search. A further investigation of these properties is beyond the scope of this thesis, and the reader is referred to [3] for a rigorous analysis of hyperparameter selection methods.

2.9 Layers

The most important layers in convolutional neural networks are the fully connected, the convolutional, and the pooling layer, all of which are explained in detail in the following sections. In a typical modern architecture, convolutional layers are placed close to the input image, whereas fully connected layers perform the high-level reasoning further down the architecture towards the loss function. Pooling layers are inserted after convolutions with the main purpose of lowering the spatial extent

of the filters, and thus the amount of learnable parameters. A high-level overview of a convolutional neural network architecture is presented in figure 2.7.

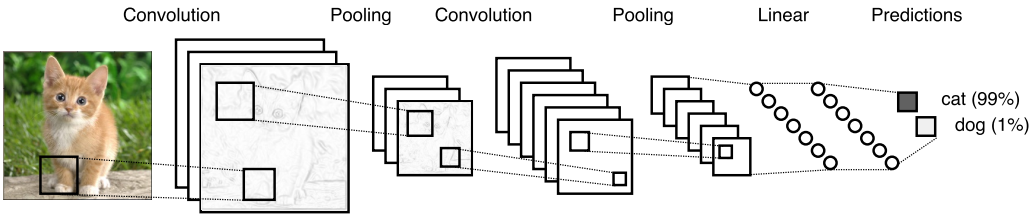
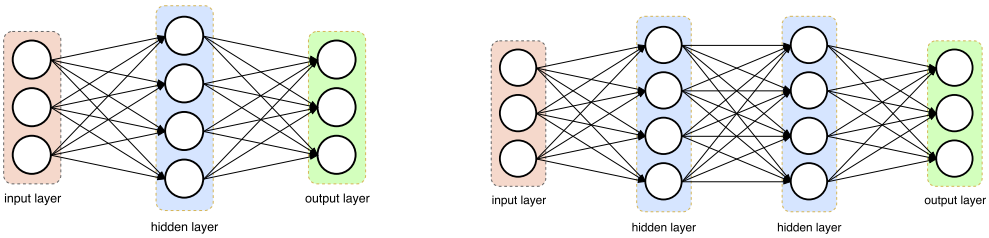


Figure 2.7: High-level overview of a convolutional neural network architecture.

2.9.1 Fully connected layer

The fully connected layer is a synonym often used in the convolutional network literature and is equivalent to a hidden layer in a regular artificial network. Is is sometimes referred to as *linear* or *affine* layer. Intuitively, the fully connected layer is responsible for the high-level reasoning in a convolutional neural network and is therefore typically inserted after the convolutional layers (section 2.9.2). The neurons have full connections to all activations in the previous layer, as seen in a regular artificial neural network (figure 2.8).



(a) One hidden layer

(b) Two hidden layers

Figure 2.8: Fully connected layers in an artificial neural network

Forward pass

The activations of a fully connected layer can be computed with a dot product of the weights with the previous layer activations followed by a bias offset as in

$$\boldsymbol{x}^{(\ell)} = (\boldsymbol{w}^{(\ell)})^T \boldsymbol{x}^{(\ell-1)} + \boldsymbol{b}^{(\ell)} \tag{2.26}$$

where $\boldsymbol{x}^{(\ell-1)}$ denotes the activations of the previous layer and $\boldsymbol{x}^{(\ell)}$, $\boldsymbol{w}^{(\ell)}$, and $\boldsymbol{b}^{(\ell)}$ denote the activations, weights, and biases of the current layer, respectively.

Backward pass

During the backward pass, the gradient with respect to the weights and biases are computed as in

$$\begin{aligned} \nabla_{\boldsymbol{w}^{(\ell)}} \mathcal{L} &= (\boldsymbol{x}^{(\ell)})^T \left(\nabla_{\boldsymbol{x}^{(\ell+1)}} \mathcal{L}\right) \\ \nabla_{\boldsymbol{b}^{(\ell)}} \mathcal{L} &= \sum_{i=1}^{n} \left(\nabla_{\boldsymbol{x}^{(\ell+1)}} \mathcal{L}\right)_i^T \end{aligned} \tag{2.27}$$

where $\nabla_{\boldsymbol{x}^{(\ell+1)}}$ denotes the upstream derivatives.

Hyperparameters

The only hyperparameter in a fully connected layer is the number of output neurons the input connects to, i.e. how many learnable parameters connect the input to the output.

Limitations

The main limitation of the fully connected layer is the assumption that each input feature, i.e. pixel in the image, is completely independent of neighboring pixels and contributes equally to the predictive performance. However, pixels that are close together have the tendency to be highly correlated and thus the spatial structure of images has to be taken into account. Additionally, fully connected layers do not scale well to high dimensional data such as images since each pixel of the input has to be connected to the layer's output with a learnable parameter.

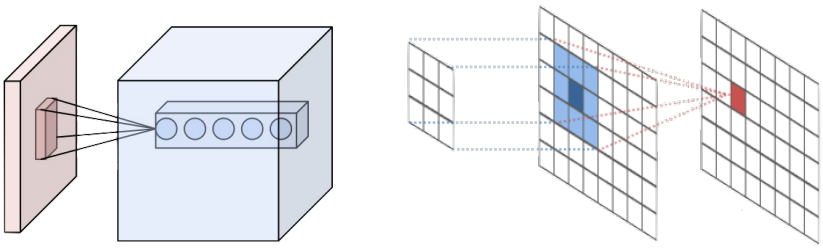
2.9.2 Convolutional layer

The convolutional layer is the core building block of a convolutional neural network and aims to resolve the limitations of fully connected layers (section 2.9.1) by making geometric assumptions about the input data. The layer's parameters consist of a set of learnable filters or kernels, which have a small receptive field, but extend through the full depth of the input volume (figure 2.9). The architecture found in convolutional layers – or convolutional networks in general – is loosely based on the complex arrangement of cells in the mammalian brain's visual cortex [27, 26, 25].

Forward pass

During the forward pass, the learnable filters are convolved with the input volume. The intuition behind the discrete convolution operation is to slide a filter over the input volume at different overlapping spatial locations as in

$$\boldsymbol{x}_f^{(\ell)} = \sum_{u,v} \boldsymbol{x}_{uv}^{(\ell-1)} * \boldsymbol{w}_f^{(\ell)} + \boldsymbol{b}_f^{(\ell)} \tag{2.28}$$



(a) 3D (b) 2D

Figure 2.9: Input, filter, and output of a convolution in 2D and 3D [12]

where $\boldsymbol{x}_f^{(\ell)}$ represents the current layer's output for a given filter f, $\boldsymbol{x}^{(\ell-1)}$ the output of the previous layer, and the spatial extent of the filter in horizontal and vertical direction is given by u and v.

Backward pass

During the backward pass, we calculate the partial derivatives of the loss function with respect to the weights and biases of the respective layer as in

$$\begin{aligned} \nabla_{\boldsymbol{w}_f^{(\ell)}} \mathcal{L} &= \sum_{u,v} \left(\nabla_{\boldsymbol{x}_f^{(\ell+1)}} \mathcal{L} \right)_{uv} (\boldsymbol{x}_{uv}^{(\ell)} * \hat{\boldsymbol{w}}_f^{(\ell)}) \\ \nabla_{\boldsymbol{b}_f^{(\ell)}} \mathcal{L} &= \sum_{u,v} \left(\nabla_{\boldsymbol{x}_f^{(\ell+1)}} \mathcal{L} \right)_{uv} \end{aligned} \tag{2.29}$$

where, $\hat{\boldsymbol{w}}$ denotes a spatially flipped filter in order to compute the cross-correlation rather than a convolution.

Hyperparameters

The hyperparameters of a convolutional layer are its spatial filter size, depth, stride, and padding. They have to be chosen carefully in order to generate a desired output.

Filter size The filter size corresponds to the spatial extent (width and height) of the filters that are convolved with the input image at different spatial locations. Generally, the filters are quadratic, i.e. have the same width and height, and all filters of a particular convolutional layer have the same filter size. The filter size is heavily dependent on the spatial extent of the input images themselves, and we generally see larger filters for larger input images. In recent publications, however, the trend is to replace larger filters with a series of small filters, which is motivated by a reduction of learnable parameters and substantial speed improvements [54, 20, 58].

Depth The depth of the output volume controls the number of learnable filters that connect to the same region of the input volume. All of these filters will learn

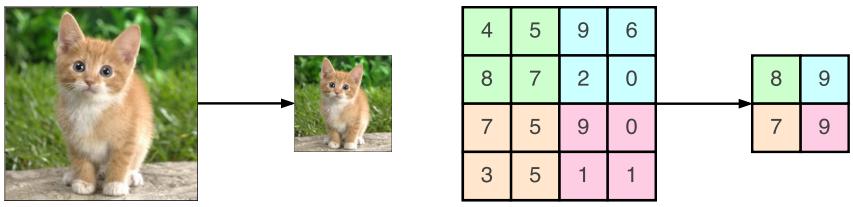
to activate for different features of the input. For example, if the first convolutional layer takes the raw image as input, then different filters along the depth dimension may activate in the presence of various oriented edges, or blobs of color, for instance.

Stride The stride parameter controls how depth columns around the spatial dimensions are allocated. When the stride is one, a new depth column of activations is allocated in positions only one spatial unit apart. This leads to heavily overlapping receptive fields between the columns, and also to large output volume. Conversely, if higher strides are used then the receptive fields will overlap less and the resulting output volume will have smaller dimensions spatially.

Padding The padding parameter allows to control the spatial size of the output volumes by extending the input volume with zeros at its outer edges. In particular, sometimes it is desirable to exactly preserve the spatial size of the input volume. In order to perform padding that is more sensitive to the content of the original image, two commonly used methods are reflection and repetition padding. Reflection padding mirrors the border content of the image outwards, whereas repetition padding fills the padded areas with the same values as the outer edges in the original image.

2.9.3 Pooling layer

Another important concept of convolutional networks is pooling, which is a form of non-linear downsampling. The pooling layer partitions the input volume into a set of non-overlapping rectangles and, for each subregion, outputs the maximum activation, hence the name max-pooling (figure 2.10). Another common pooling operation is average pooling, which computes the mean of the activations in the previous layer rather than the maximum. The function of the pooling layer is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. It is common practice to periodically insert a pooling layer in between successive convolutional layers.



(a) Global (b) Local

Figure 2.10: Max-pooling layer

Forward pass

During the forward pass, the maximum of non-overlapping regions of the previous activations is computed as in

$$\boldsymbol{x}^{(\ell)} = \max_{u,v} \left(\boldsymbol{x}^{(\ell-1)} \right)_{uv} \tag{2.30}$$

where u and v denote the spatial extent of the non-overlapping regions in width and height.

Backward pass

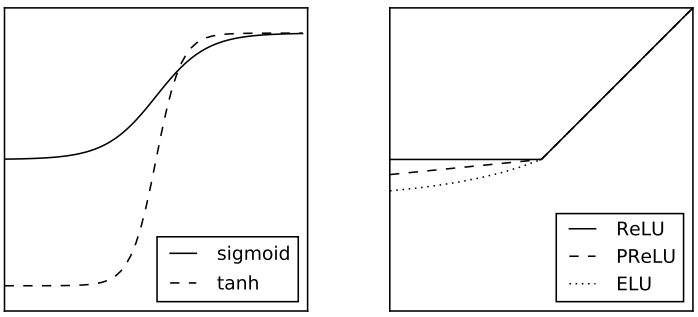
Since the pooling layer does not have any learnable parameters, the backward pass is merely an upsampling operation of the upstream derivatives. In case of the max-pooling operation, it is common practice to keep track of the index of the maximum activation so that the gradient can be routed towards its origin during backpropagation.

Hyperparameters

The hyperparameters of the pooling layer are its stride and filter size. Since they have to be chosen in accordance to each other, they can be interpreted as the amount of downsampling to be performed.

2.10 Activation functions

An activation function, sometimes called *nonlinearity* in the convolution neural network context, takes a single number and performs a fixed mathematical operation on it. There are several activation functions in use today, the most common of which are explained in detail in the following sections. Figure 2.11 visualizes the most common activation functions.



(a) Saturating nonlinearities (b) Rectified nonlinearities

Figure 2.11: The most common saturating and rectified activation functions.

2.10.1 Saturating activation functions

Both the sigmoid and the hyperbolic tangent activation function can be seen as saturating nonlinearities (figure 2.11a) [36, 24]. In recent years, they have largely fallen out of favor and have been replaced with so-called rectifiers. However, understanding the difficulties that arise from using such saturating nonlinearities is crucial for motivating rectified activation functions.

Sigmoid

The sigmoid activation function $\sigma(x)$ squashes a real-valued number into the range between zero and one as in

$$\sigma(x) = \frac{1}{(1 + e^{-x})} \tag{2.31}$$

It has seen frequent use historically since it has a nice interpretation as the firing rate of a neuron, from not firing at all to fully-saturated firing at an assumed maximum frequency. In practice, the sigmoid activation function has recently fallen out of favor and is rarely used since it has two major drawbacks.

A very undesirable property of the sigmoid function is that its activations saturate at either tail of zero or one and the gradient at these regions is very close to zero. During backpropagation, the local gradient will be multiplied by the gradient of this layer's output for the overall loss. Therefore, if the local gradient is very small, it will effectively diminish the gradient and almost no signal will flow through the unit to its weights and recursively to its data. This is called the *vanishing gradient problem* [24].

Another undesirable property of the sigmoid activation function is that its output is non-zero centered. This has implications on the dynamics of the network during gradient descent since the data flowing into a neuron is always positive. The gradient on the weights during backpropagation will either become all-positive or all-negative, depending on the gradient of the whole expression. This could introduce undesirable zig-zagging dynamics in the updates of the weights [38].

Hyperbolic tangent

The hyperbolic tangent activation function $\tanh(x)$ squashes a real-valued number to the range between negative one and one as in

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{2.32}$$

Like the sigmoid nonlinearity, its activations saturate, but its output is zero centered. Therefore, in practice, the hyperbolic tangent is always preferred to the sigmoid nonlinearity [38]. Note that the hyperbolic tangent function is simply a scaled sigmoid function that is zero centered.

2.10.2 Rectified activation functions

Modern activation functions such as rectifiers have become increasingly popular since they were found to greatly accelerate the convergence of stochastic gradient descent compared to saturating nonlinearities such as sigmoid or hyperbolic tangent [15].

Rectified Linear Unit (ReLU)

The ReLU activation function is linear and simply threshold at zero and can therefore be expressed as

$$\mathrm{ReLU}(x) = \max(0, x) \tag{2.33}$$

In practice, ReLU units have one major drawback that arises from their simplicity. They can be very fragile during training because of their zero gradient when $x < 0$. A large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, the gradient flowing through the unit will forever be zero, i.e. the unit can irreversibly die during training. This phenomenon is referred to as *dead neurons* in the neural network context.

Parametric and leaky ReLU

The parametric and leaky ReLU activation functions are one attempt to fix the dead neuron problem. The parametric ReLU activation function (PReLU) is can be seen as a generalization of the leaky ReLU (LReLU). Instead of the function being zero when $x < 0$, a leaky or parametric ReLU will have a small negative slope as in

$$\mathrm{PReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{otherwise} \end{cases} \tag{2.34}$$

where α is a small constant such as 0.01 for the leaky ReLU, and a learnable parameter for the parametric ReLU.

Exponential linear unit (ELU)

The exponential linear unit (ELU) [5] is similar to other rectifiers but has smooth negative values as in

$$\mathrm{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases} \tag{2.35}$$

where α is a hyperparameter that controls the value to which an ELU saturates for negative inputs.

The exponential linear unit allows the network to push mean unit activations closer to zero like batch normalization [28] but with lower computational complexity. The authors argue that ELUs significantly outperform ReLU networks with batch normalization, while batch normalization does not improve ELU networks.

2.11 Loss functions

The loss layer is one of the essential parts of any neural network. Since we are in a supervised learning context, our network computes predictions for any given input image, which we in turn compare to its corresponding ground truth label. We measure the "unhappiness" with the predictions that our network produces using the current parameters with a loss function, sometimes also referred to as *cost function*, *objective function*, or *criterion*. In a multi-class classification context, it is useful to regard the output from the last fully connected layer as class scores denoted by $\boldsymbol{s}$. For the i-th training example, we thus have input $\boldsymbol{x}_i$, ground truth label $\boldsymbol{y}_i$ and the corresponding score $\boldsymbol{s}_i$. There are several ways to define the loss function, the most important of which are discussed in further detail in the following sections.

Geometrically, the loss function of a neural network can be interpreted as a landscape as depicted in figure 2.12. Its height is determined by the parameters $\boldsymbol{\theta}$. In this two dimensional example, we are trying to move from the areas with a high loss (red) into the areas with a low loss (blue) by incrementally adjusting the parameters using backpropagation. It is important to note that the global minimum might not be unique and that there may exist several local minima [38].

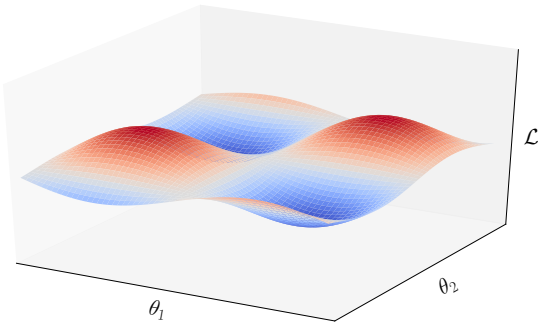


Figure 2.12: Geometrical interpretation of a loss landscape in two dimensions.

2.11.1 Hinge loss

A commonly used loss function for the image classification task in neural networks is the multi-class hinge loss, also referred to as support vector machine (SVM) loss.

The hinge loss is set up such that the correct class for each input has a higher score than the incorrect classes by some fixed margin. Mathematically, the multi-class hinge loss can be formalized as

$$\mathcal{L}_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \delta) \tag{2.36}$$

where $\mathcal{L}_i$ denotes the loss for the i-th training example and δ is the fixed margin.

We can see that the hinge loss prefers the score of the correct class y_i to be larger than the incorrect class scores by at least a margin δ. If this is not the case, loss is accumulated. Another related loss function is the squared hinge loss, which penalizes violated margins quadratically instead of linearly, and thus more strongly. The squared hinge loss can be formalized as

$$\mathcal{L}_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \delta)^2 \tag{2.37}$$

There are various alternatives of the multi-class hinge-loss such as the one-vs-all formulation, which trains an independent binary SVM for each class, or the structured SVM, which maximizes the margin between the score of the correct class and the score of the highest-scoring incorrect runner-up class. However, a discussion of these alternative formulations is beyond scope of this thesis.

2.11.2 Cross entropy loss

The most popular loss function for image classification in neural networks is the cross entropy loss generalized to multiple classes via the softmax function and the negative log likelihood. Mathematically, the cross entropy loss has the form

$$\mathcal{L}_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_{j=1}^{n} e^{s_j}}\right) = -s_{y_i} + \log \sum_{j=1}^{n} e^{s_j} \tag{2.38}$$

where $\mathcal{L}_i$ denotes the loss for the i-th training example.

Both hinge and cross entropy loss usually result in comparable classification performance. However, unlike the hinge loss, which treats the outputs as uncalibrated – and possibly difficult to interpret – scores for each class, the cross entropy loss results in normalized class probabilities. In addition, the cross entropy loss has a rigorous interpretation in the domains of probability and information theory, which is why it is the preferred choice of loss function for classification tasks in practice.

Probabilistic interpretation

In the probabilistic interpretation, the negative log likelihood of the correct class is minimized, which can be seen as performing maximum likelihood estimation (MLE). If we take L_2 weight regularization into account, we can interpret the loss function as having a gaussian prior on the weights and we are thus performing maximum a

posteriori (MAP) estimation. The cross entropy loss can be seen as the normalized probability assigned to the correct class label given the input as in

$$P(y_i|x_i) = \frac{e^{s_{y_i}}}{\sum_{j=1}^{n} e^{s_j}} \tag{2.39}$$

where x_i is an input with the corresponding class label y_i and $P(y_i|x_i)$ is parametrized by the model parameters.

Information theory interpretation

In the information theory interpretation, the cross entropy between a true distribution p and an estimated distribution q is defined as

$$H(p,q) = \underbrace{-\sum_{x} p(x) \log q(x)}_{\text{cross entropy}} = \underbrace{H(p)}_{\text{entropy}} + \underbrace{D_{\mathrm{KL}}(p||q)}_{\text{KL divergence}} \tag{2.40}$$

The classifier is thus minimizing the cross entropy between the estimated class probabilities and the true distribution, which in this interpretation is the distribution where all probability mass is on the correct class, i.e. a vector of all zeros with a single one at the y_i-th position. In the literature, this is referred to as one-hot encoding. Moreover, since the cross entropy can be written in terms of entropy and the Kullback-Leibler divergence, it is equivalent to minimizing the KL divergence – a difference measure – between two distributions. Thus, the cross entropy loss wants the predicted distribution to have all of its mass on the correct answer. It has to be noted that the KL divergence is often used in practice but does not correspond to a true distance metric since it does not obey the triangle inequality, and in general $D_{\mathrm{KL}}(p||q)$ does not equal $D_{\mathrm{KL}}(q||p)$.

2.12 Regularization

Regularization is a very important technique to prevent overfitting in machine learning problems. From a theoretical point of view, the generalization error can be broken down into two distinct origins, namely error due to *bias*, and error due to *variance*. These terms are loosely related to – but not to be confused with – the bias parameters of the network and the statistical measure of variance, respectively.

The bias is a measure of how much the network output, averaged over all possible datasets differs from the desired function. The variance is a measure of how much the network output varies between datasets. Early in training, the bias is large because the network output is far from the desired function. The variance is very small since the data has had little influence on the model parameters yet. Later in training, however, the bias is small because the network has learned the underlying representation and has the ability to approximate the desired function. If trained too long, and assuming the model has enough representational power, the network

will also have learned the noise specific to that dataset, which is referred to as overfitting. In case of overfitting, we have poor generalization, and the variance will be large because the noise varies between datasets. It can be shown that the minimum total error will occur when the sum of bias and variance are minimal.

In a neural network setting, the loss function is not convex and may thus contain many different local minima. During training, we want to reach a local minimum that explains the data in the simplest possible way according to Occam's razor, thus having a high chance of generalization.

2.12.1 Early stopping

Early stopping is the easiest and most effective method for regularizing the solution and to aid generalization performance. Simply put, the validation accuracy on the held-out validation set is continuously monitored and the training is stopped once the accuracy stops improving. In practice, one can save the best-performing model parameters in addition to the current parameters and fall back on the saved one once further improvements seem unlikely.

2.12.2 Weight regularization

A common practice is to introduce an additional term to the loss function such that the total loss is a combination of data loss and regularization loss as in

$$\mathcal{L}(\boldsymbol{w}) = \underbrace{\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}_i}_{\text{data loss}} + \underbrace{\lambda R(\boldsymbol{w})}_{\substack{\text{regularization} \\ \text{loss}}} \tag{2.41}$$

where $\mathcal{L}$ can be any loss function, $R(\boldsymbol{w})$ is the regularization penalty and λ is a hyperparameter that controls the regularization strength.

The intuition behind weight regularization is therefore to prefer smaller weights, and thus the local minima which have a simpler solution. In a neural network setting, this technique is also referred to as *weight decay*. In practice, regularization is only applied to the weights of the network, not its biases. This stems from the fact that the biases do not interact with the data in a multiplicative fashion, and therefore do not have much influence on the loss.

The regularization penalty can be defined in a number of ways, the most popular of which are discussed below.

Lasso regularization

Lasso or L_1 regularization encourages zero weights and therefore sparsity. It is often encountered and computes the L_1 norm of the weights, i.e. the sum of absolute values as in

$$R_{\text{Lasso}}(\boldsymbol{w}) = ||\boldsymbol{w}||_1 = \sum_{i=1}^{n}\sum_{j=1}^{m}|w_{ij}| \tag{2.42}$$

Ridge regularization

Ridge or L_2 regularization encourages small weights. It is the most popular choice of weight regularization in practice and computes the L_2 norm of the weights, i.e. the sum of squares as in

$$R_{\text{Ridge}}(\boldsymbol{w}) = ||\boldsymbol{w}||_2^2 = \sum_{i=1}^{n}\sum_{j=1}^{m}w_{ij}^2 \tag{2.43}$$

Elastic net regularization

Elastic net regularization combines L_1 and L_2 regularization. Elastic net tends to have a grouping effect, where correlated input features are assigned equal weights. It is commonly used in practice and is implemented in many machine learning libraries. Elastic net regularization can be formalized as

$$R_{\text{Elastic}}(\boldsymbol{w}) = \alpha||\boldsymbol{w}||_1 + (1-\alpha)||w||_2^2 \tag{2.44}$$

where the amount of L_1 or L_2 regularization can be adjusted via the hyperparameter $\alpha \in [0, 1]$.

A two dimensional geometric representation of the weight regularization methods is depicted in figure 2.13.

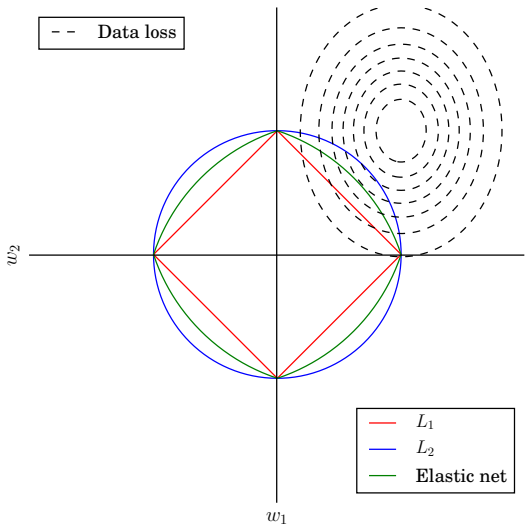
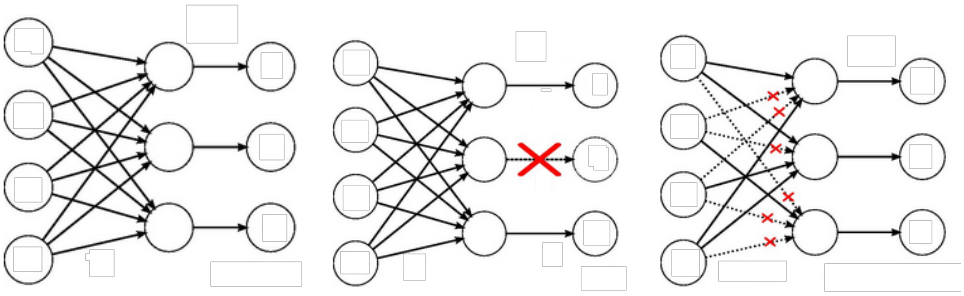


Figure 2.13: Geometrical interpretation of weight regularization methods.

2.12.3 Dropout and dropconnect

Dropout (figure 2.14b) [23, 64, 55, 66] is a radically different technique for regularization. Unlike weight regularization, dropout does not rely on modifying the loss function but the network itself. The key idea is to randomly drop units from the neural network during training and thus preventing the co-adaptation of features. At each training stage, individual nodes are either dropped out of the network with probability $1-p$ or kept with probability p, such that a reduced network is left and individual activations cannot rely on other activations to be present simultaneously.

Dropconnect (figure 2.14c) [65] can be seen as a generalization of the dropout mechanism. Instead of randomly dropping neurons during training, only the connections between them are set to zero and therefore disregarded during backpropagation. Dropconnect is similar to dropout as it introduces dynamic sparsity within the model, but differs in that the sparsity is on the weights, rather than the output of a layer. A visual comparison of dropout and dropconnect in a fully connected layer is presented in figure 2.14.



(a) Fully connected layer (b) Dropout layer (c) Dropconnect layer

Figure 2.14: Comparison of dropout and dropconnect layers [65]

2.12.4 Data augmentation

A general way to reduce overfitting is to introduce more training data. In many cases, however, it is both practical and beneficial to systematically alter the existing training data to generate more examples while preserving the label. In the case of image classification, the input images can be transformed in the following ways: horizontal and vertical flipping, cropping, scaling, translating, rotating, color jittering or shifting, and contrast and brightness changes.

Chapter 3

Batch normalization

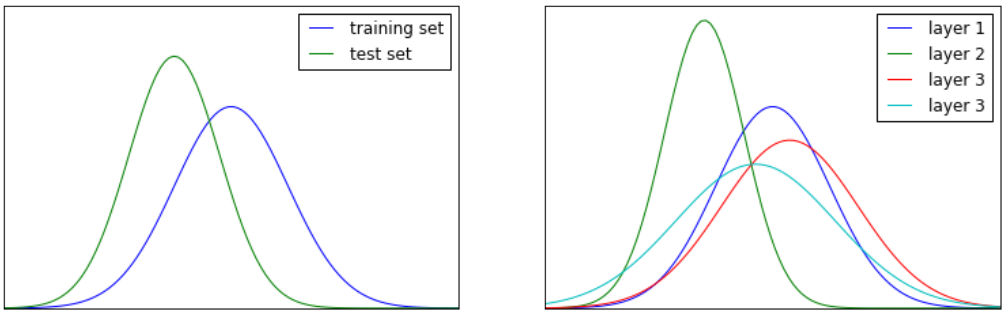
Batch normalization [28] is a recently popularized method for accelerating deep network training by making data standardization an integral part of the network architecture. Batch normalization can be seen as yet another layer that can be inserted into the model architecture, just like the fully connected or convolutional layer. It provides a definition for feed-forwarding the input and computing the gradients with respect to the parameters and its own input via a backward pass. In practice, batch normalization layers are inserted after a convolutional or fully connected layer, but before the outputs are fed into an activation function. For convolutional layers, the different elements of the same feature map – i.e. the activations – at different locations are normalized in the same way in order to obey the convolutional property. Thus, all activations in a mini-batch are normalized over all locations, rather than per activation.

The authors of batch normalization claim that the *internal covariate shift* is the major reason why deep architectures have been notoriously slow to train. This stems from the fact that deep networks do not only have to learn a new representation at each layer, but also have to account for the change in their distribution.

The *covariate shift* in general is a known problem in the machine learning community and frequently occurs in real-world problems [52]. A common covariate shift problem is the difference in the distribution of the training and test set which can lead to suboptimal generalization performance (figure 3.1a). This problem is usually handled with a standardization or whitening preprocessing step (section 2.3). However, especially the whitening operation is computationally expensive and thus impractical in an online setting, especially if the covariate shift occurs throughout different layers [28].

The *internal covariate shift* is the phenomenon where the distribution of network activations change across layers due to the change in network parameters during training (figure 3.1b). Ideally, each layer should be transformed into a space where they have the same distribution but the functional relationship stays the same. In order to avoid costly calculations of covariance matrices to decorrelate and whiten the data at every layer and step, we normalize the distribution of each input feature

in each layer across each mini-batch to have zero mean and a standard deviation of one.



(a) Covariate shift (b) Internal covariate shift

Figure 3.1: Covariate shift vs. internal covariate shift

3.1 Forward pass

During the forward pass, we compute the mini-batch mean and variance. With these mini-batch statistics, we normalize the data by subtracting the mean and dividing by the standard deviation. Finally, we scale and shift the data with the learned scale and shift parameters. The batch normalization forward pass f_{BN} can be described mathematically as

$$\begin{aligned} \mu_{\mathcal{B}} &= \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i^{(\ell-1)} \\ \sigma_{\mathcal{B}}^2 &= \frac{1}{n}\sum_{i=1}^{n} (\boldsymbol{x}_i^{(\ell-1)} - \mu_{\mathcal{B}})^2 \\ \hat{\boldsymbol{x}}^{(\ell-1)} &= \frac{\boldsymbol{x}^{(\ell-1)} - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \\ \boldsymbol{x}^{(\ell)} &= \boldsymbol{\gamma}^{(\ell)} \hat{\boldsymbol{x}}^{(\ell-1)} + \boldsymbol{\beta}^{(\ell)} \end{aligned} \tag{3.1}$$

where $\mu_{\mathcal{B}}$ is the batch mean and $\sigma_{\mathcal{B}}^2$ the batch variance, respectively. The learned scale and shift parameters are denoted by $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$, respectively. For clarity, we describe the batch normalization procedure per activation and omit the corresponding indices.

Since normalization is a differentiable transform, we can propagate errors into these learned parameters and are thus able to restore the representational power of the network by learning the identity transform. Conversely, by learning scale and shift parameters that are identical to the corresponding batch statistics, the batch normalization transform would have no effect on the network, if that was the optimal operation to perform. At test time, the batch mean and variance are

replaced by the respective population statistics since the input does not depend on other samples from a mini-batch. Another popular method is to keep running averages of the batch statistics during training and to use these to compute the network output at test time. At test time, the batch normalization transform can be expressed as

$$\begin{aligned} \hat{\boldsymbol{x}}^{(\ell-1)} &= \frac{\boldsymbol{x}^{(\ell-1)} - \mu_{\mathcal{D}}}{\sqrt{\sigma_{\mathcal{D}}^2 + \epsilon}} \\ \boldsymbol{x}_i^{(\ell)} &= \boldsymbol{\gamma}^{(\ell)} \hat{\boldsymbol{x}}_i^{(\ell-1)} + \boldsymbol{\beta}^{(\ell)} \end{aligned} \tag{3.2}$$

where $\mu_{\mathcal{D}}$ and $\sigma_{\mathcal{D}}^2$ denote the population mean and variance, rather than the batch statistics, respectively.

3.2 Backward pass

Since normalization is a differentiable operation, the backward pass can be computed as

$$\begin{aligned} \nabla_{\boldsymbol{\gamma}^{(\ell)}} \mathcal{L} &= \sum_{i=1}^{n} \left(\nabla_{\boldsymbol{x}^{(\ell+1)}} \mathcal{L}\right)_i \cdot \hat{\boldsymbol{x}}_i^{(\ell)} \\ \nabla_{\boldsymbol{\beta}^{(\ell)}} \mathcal{L} &= \sum_{i=1}^{n} \left(\nabla_{\boldsymbol{x}^{(\ell+1)}} \mathcal{L}\right)_i \end{aligned} \tag{3.3}$$

3.3 Properties

One of the most intriguing properties of batch normalized networks is its invariance to parameter scale. The bias term can be omitted since the effect will be cancelled in the subsequent mean subtraction. The scale invariance property in a batch normalized fully connected layer can be formalized as

$$f_{\mathrm{BN}}((\alpha \boldsymbol{w})^T \boldsymbol{x} + \boldsymbol{b}) = f_{\mathrm{BN}}(\boldsymbol{w}^T \boldsymbol{x}) \tag{3.4}$$

where α denotes the scale parameter.

In a batch normalized fully connected layer, we can further show that the parameter scale does not affect the layer Jacobian and therefore the gradient propagation. Batch normalization layers also have a stabilizing property since larger weights lead to smaller gradients because of their inversely proportional relationship. Both of these properties can be formalized as

$$\begin{aligned} \nabla_{\boldsymbol{x}} f_{\mathrm{BN}}((\alpha \boldsymbol{w})^T \boldsymbol{x}) &= \nabla_{\boldsymbol{x}} f_{\mathrm{BN}}(\boldsymbol{w}^T \boldsymbol{x}) \\ \nabla_{(\alpha \boldsymbol{w})} f_{\mathrm{BN}}((\alpha \boldsymbol{w})^T \boldsymbol{x}) &= \frac{1}{\alpha} \nabla_{\boldsymbol{w}} f_{\mathrm{BN}}(\boldsymbol{w}^T \boldsymbol{x}) \end{aligned} \tag{3.5}$$

where α denotes the scale parameter.

Another observation is that batch normalization may lead the layer Jacobian to have singular values close to one, which is known to be beneficial for training [51]. Assume two consecutive layers with normalized inputs and their transformation $f(\boldsymbol{x}) = \boldsymbol{y}$. If we assume that $\boldsymbol{x}$ and $\boldsymbol{y}$ are normally distributed and decorrelated, and that $f(\boldsymbol{x}) \approx J\boldsymbol{x}$ is a linear transformation given the model parameters, then both $\boldsymbol{x}$ and $\boldsymbol{y}$ have unit covariance. Thus, $I = JJ^T$, and all singular values of J are equal to one, which preserves the gradient magnitudes during backpropagation.

3.4 Hyperparameters

Simply adding batch normalization to a network does not take full advantage of the method and a few alterations can be made with respect to the model hyperparameters. These include – but are not limited to – the following: increasing the learning rate (section 2.6), removing dropout (section 2.12.3), reducing weight regularization (section 2.12.2), accelerating the learning rate decay (section 2.8.2), reducing photometric distortions while preprocessing the data (section 2.12.4), and shuffling training examples more thoroughly (section 2.6.3).

3.5 Motivation

As mentioned before, deep architectures involve the composition of several computational layers. The gradient computed with backpropagation tells the network by how much each parameter should be updated, under the assumption that the other layers do *not* change. If we update all the layers simultaneously, as it is common when using gradient descent, unexpected fluctuations can happen since we perform updates assuming the other functional dependencies remain constant.

Assume we have the following network with ℓ layers that uses only one weight per layer. For simplicity, we are not using any nonlinearities or biases. The network architecture can be described as

$$f(x, \boldsymbol{w}) = x \cdot w_1 \cdot w_2 \cdot \ldots \cdot w_\ell = \hat{y} \tag{3.6}$$

Clearly, the output $\hat{y}$ is a linear function of the input x, but a nonlinear function of the weights w_i. Further suppose that our loss function has computed a gradient of 1, so we wish to decrease $\hat{y}$ slightly. The backpropagation algorithm then computes the gradient as in

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \cdot \nabla_{\boldsymbol{w}} f(x) \tag{3.7}$$

However, the actual update will include second-order and third-order effects, up to order ℓ. The new value of $\hat{y}$ is then given by the expression

$$f(x, \boldsymbol{w}) = x \cdot \left(w_1 - \eta \cdot \frac{\partial f(x, \boldsymbol{w})}{\partial w_1}\right) \cdot \ldots \cdot \left(w_\ell - \eta \cdot \frac{\partial f(x, \boldsymbol{w})}{\partial w_\ell}\right) = \hat{y} \tag{3.8}$$

This property makes it very hard to choose an appropriate learning rate since the effects of an update to the parameters for one layer depend so strongly on all of the other layers. As we have seen in section 2.7.3, second-order optimization methods are already very expensive so taking even higher-order interactions into account seems hopeless.

Batch normalization alleviates some of the hopelessness by assuming that if $\boldsymbol{x}$ is drawn from a unit normal distribution, all the downstream layers will be normally distributed because the intermediate transformations are linear. However, the intermediate layers will no longer have zero mean and unit variance. After applying batch normalization, all the intermediate layers will have restored their zero mean and unit variance property. Learning this model now becomes much simpler since the weights at other layers simply do not have an effect in most cases since their activations are continuously renormalized [16].

Chapter 4

Related work

Feature normalization is a standard preprocessing step applied to various machine learning problems including neural networks. The motivation behind it, on the one hand, is that the loss function does not behave as expected without it. For instance, in the majority of classification tasks, the loss layer calculates a distance between the model predictions and the ground truth labels. If one of the features has a broader range of values, the loss will be governed by that particular feature, and therefore the range of all features should be normalized so that each one contributes approximately proportionally to the final loss. Empirically, stochastic gradient descent has been found to converge much faster with feature normalization than without it. Batch normalization can be regarded as a normalization preprocessing step with the difference that it is applied at various stages inside of the network rather than before the data is fed into the network. In the literature, batch normalization layers have been applied to various neural network architectures with high levels of success [63, 62, 29, 44, 10].

The following sections are divided into *before* and *beyond* batch normalization. The first section highlights the most important work in feature normalization and resembles the state-of-the-art in standardization procedures before batch normalization was published. The second section features work that specifically takes batch normalization as a basis for further improvements.

4.1 Before batch normalization

LeCun et al. [38] show that preprocessing the data by subtracting the mean, normalizing the variance, and decorrelating the input has various beneficial effects for backpropagation (section 2.3). Subtracting the mean helps the learning process because non-zero mean in the input data creates a large eigenvalue, and thus a high condition number. The condition number measures how much the output value of a function can change for small change in the input argument. By subtracting the mean, we create a well-conditioned problem, i.e. a problem with a lower condition number. Furthermore, variance normalization should be applied since inputs

that have a large variation in spread along different directions of the input space will have a larger condition number and result in slow learning. Finally, decorrelating the input data usually causes the eigenvalues of the Hessian to be rotated towards the coordinate axes and thus decouples the weight updates. The authors further conjecture that adaptive optimization methods, i.e. adaptive learning rates for different weights, are beneficial for backpropagation.

Glorot et al. [14] shine light onto the difficulty of training deep neural networks and they show the superiority of deeper architectures as opposed to shallow ones. Their main contribution is the Xavier initialization scheme (section 2.4.2) which results in substantially faster convergence for deep architectures. In their work, they empirically evaluate the proposed initialization scheme with an extensive gradient propagation study that exposes the internal covariate shift by visualizing the activation distribution across different layers. Conducting such a gradient propagation study for batch normalized networks might be useful to verify if and to what extend the internal covariate shift is an issue when training deep neural networks.

Krizhevsky et al. [33] propose a method called local response normalization which is inspired by computational neuroscience and acts as a form of lateral inhibition, i.e. the capacity of an excited neuron to reduce the activity of its neighbors. The technique models real neurons and creates competition for big activities amongst layer outputs computed using different kernels. In contrast to batch normalization, local response localization is applied after the nonlinearity and sums over an arbitrary number of adjacent kernel maps. The method has proven to be effective for certain tasks but is neither thoroughly evaluated nor motivated. Local response normalization is tightly connected to the local contrast normalization method proposed by Jarrett et al. [30].

Gülçehre et al. [18] propose a standardization layer that bears significant resemblance of a batch normalization layer, except the two methods are motivated by very different goals and perform different tasks. The goal of batch normalization is to achieve a stable distribution of activation values throughout training, whereas Gülçehre et al. apply the standardization layer to the output of the nonlinearity, which results in sparser activations. Other notable differences include the learned scale and shift parameters that allow batch normalization layers to represent the identity transform, which the proposed method is lacking.

Clevert et al. [5] introduce a new type of nonlinearity, namely the exponential linear unit (ELU) which supposedly speeds up leaning in neural networks and leads to higher classification accuracies. They conjecture that ELUs alleviate the vanishing gradient problem via the identity of positive values but improve learning characteristics due to their smooth negative values, which have the effect of pushing the mean unit activations closer to zero as in batch normalization. They compare their nonlinearity to parametric rectifiers such as the leaky and parametric ReLU and empirically achieve better results. From an intuitive standpoint, they argue that leaky and parametric ReLU do not ensure a noise robust deactivation state whereas ELUs saturate to negative values and decrease the forward propagated variation of information. Therefore, ELUs have the effect of coding the degree of presence

of particular phenomena in the input, while they do not quantitatively model the degree of their absence. Furthermore, they claim that ELU networks significantly outperform batch normalized networks, while batch normalization does not improve ELU networks.

4.2 Beyond batch normalization

Devansh et al. [1] propose a parametric technique for alleviating the internal covariate shift called normalization propagation. The authors tackle two fundamental problems of batch normalization, namely the computation of batch statistics is costly and that batch normalization does not handle mini-batch stochastic gradient descent with a batch size of one. Their method uses a data-independent parametric estimate of the batch mean and standard deviation and thus does not rely on the expensive calculations of batch statistics or the batch size being greater than one. The assumption being that the network activations are normally distributed, just like in batch normalization. Their empirical results verify that normalization propagation does in fact act like batch normalization with the drawback that it heavily relies on the ReLU nonlinearity and has to be implemented separately for other activation functions.

Liao et al. [41] highlight the importance of normalization layers in deep networks with piecewise linear activation functions such as ReLU, leaky ReLU, and parametric ReLU. In their experiments on the MNIST, SVHN, CIFAR10, and CIFAR100 datasets, they found that adding batch normalization is essential not only for training networks with saturating nonlinearities, but also for piecewise linear activation functions. They argue that batch normalization improves ill-conditioning of the problem as the network depth increases. Furthermore, they show that adding batch normalization before the nonlinear activation functions is vital in order for the network training to accelerate and achieve higher accuracies.

Laurent et al. [34] experimentally show that the batch normalization procedure for deep feedforward networks cannot be easily extended to recurrent neural networks. They find that adding batch normalization to the hidden-to-hidden transitions in a recurrent network does not improve convergence speed. However, when applied to the input-to-hidden transitions, batch normalization can lead to faster convergence of the training criterion but does not seem to improve the generalization performance on both language modeling and speech recognition tasks.

Cooijmans et al. [8] propose a reparametrization of a specialized recurrent neural network called Long Short-Term Memory (LSTM) that brings the benefits of batch normalization to such recurrent models. They argue that adding batch normalization layers to both input-to-hidden and hidden-to-hidden transitions in LSTM networks improve tasks such as sequence classification, language modeling, and question answering. In their experiments, batch normalized LSTM networks lead to both faster convergence speed and improved generalization.

Salimans et al. [50] propose a reparametrization that is based on the batch

normalization procedure but instead of normalizing the activations between hidden layers, the network weights are normalized. They show that this arguably simple reparametrization does not introduce dependencies between the examples of a mini-batch and can be applied to recurrent networks including LSTM and to noise-sensitive applications such as deep reinforcement learning or generative models. In their experiments on supervised image recognition, generative modeling, and deep reinforcement learning, weight normalization provides much of the speed improvements of batch normalization, despite its simplicity.

Chapter 5

Method

In order to find a suitable methodology for our experiments, it is vital to recall the alleged effects of batch normalization, which can be broadly categorized as convergence speed and generalization performance improvements. We will make use of a toy model that can be trained quickly before moving to a more elaborate model that resembles state-of-the-art architectures. This way, we are able to rule out the types of experiments that lead to insignificant results before wasting large amounts of time in training the complex model. In general, we compare the model with batch normalization to the exact same model without batch normalization to show how the batch normalization technique in isolation affects the training behavior.

The following sections are focused on experiments with selected hyperparameters such as the type of activation function or initial learning rate. By observing the training behavior of the two models in comparison while simultaneously altering selected hyperparameters, we aim to isolate the effect of batch normalization in general and also with respect to those hyperparameters.

5.1 Activation functions

Batch normalized networks can be trained with saturating nonlinearities and gain improvements from using exponential linear unit nonlinearities.

By leaving the model fixed and only changing the activation functions, we are able to assess how different nonlinearities affect the model's training behavior. The gradients used for the individual parameter updates will provide insight into what causes possible divergence.

5.2 Initial learning rate

The authors of batch normalization claim that their method accelerates the learning process and leads to higher accuracies by enabling much higher learning rates without the risk of divergence.

In order to assess the speed and generalization improvement properties, we will systematically alter the initial learning rate of both the vanilla and batch normalized networks, while monitoring both the training loss and validation accuracy as the training progresses. From these experiments, we will gain valuable insights into the network behavior in terms of convergence speed and generalization performance. In case that a network shows divergence, i.e. is not able to reach favorable parameter states that result in accurate class predictions, we will monitor the gradients responsible for each parameter update in order to track down the source of the model instability.

Finally, we will assess the claim that batch normalized networks can be trained using saturated nonlinearities by equipping our model with sigmoid activation functions. Another experiment will compare ReLU with ELU nonlinearities in order to verify or falsify the claim that batch normalization has no beneficial effect on models with ELU nonlinearities.

5.3 Regularization

Batch normalization reduces overfitting and acts as a regularizer, in some cases eliminating the need for dropout and allowing weaker L_2 weight regularization.

The regularization improvements are assessed by simply adding dropout and/or weight regularization to the model, while again monitoring the training loss and validation accuracy over time. We will break the total loss down into data and regularization loss to assess how much of the regularization is caused by either batch normalization or the traditional regularization techniques.

5.4 Weight initialization

Batch normalization is invariant to parameter scale in the forward pass and therefore allows for less careful model initialization. During the backward pass, the parameter scale and the gradient magnitude are inversely proportional.

To asses how the scale of the weight initialization affects the training behavior, we train both the vanilla and batch normalized network with initial values drawn from distributions with different variances. As discussed before, we monitor the propagated gradients in order to assess any possible divergence we might encounter.

5.5 Batch size

A hyperparameter that is not discussed at all by [28] is the choice of batch size for individual parameter updates when using batch normalization. Since the batch normalization technique is inherently dependent on the number of examples in a batch, we argue that performing experiments with different batch sizes will give some valuable insights.

To asses how different batch sizes affect the convergence and generalization performance, we will systematically alter the amount of examples used for a parameter update. Theoretically, changing the batch size should not have an impact on the final accuracy of the classifier, but might drastically influence the speed of convergence. With smaller batch sizes, the model performs more noisy updates which can have a regularizing effect but may lead to higher loss and accuracy fluctuations during training. Additionally, the noisy updates might enable the model to escape unfavorable local minima faster and more reliably and thus learn faster. With larger batch sizes, the parameter updates are closer to the actual gradient that leads to loss reduction, which is why we expect less fluctuations in general. To summarize, interpreting the effects of batch normalization and the batch size in conjunction is a difficult task since we cannot reliably separate the regularization caused by noisy gradient descent updates from the regularizing effect of batch normalization simultaneously.

In practice, the batch size is usually chosen with regards to how much data can fit in memory and processed in parallel. Thus, the batch size hyperparameter is generally not tuned with cross validation but rather selected to match the limitations of the hardware at hand.

Chapter 6

Experimental setup

The following sections provide an overview of the experimental setup that is used in order to verify if the desirable properties of batch normalization can be empirically verified. Specifically, we present an overview of the datasets and models used for the experiments. For an overview of the hardware and software used for the experiments, the reader is referred to appendix A.

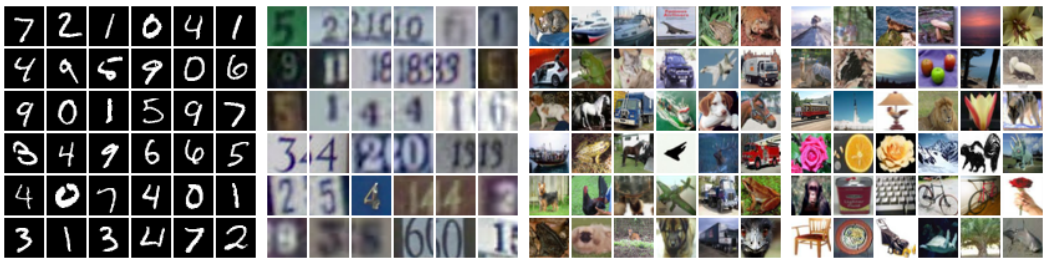
6.1 Datasets

The datasets used for the following experiments are arguably the most popular ones used for image classification tasks in the literature. Ordered by increasing difficulty to generalize, those are MNIST [39], SVHN [43], CIFAR10, and CIFAR100 [32]. They provide a good baseline for research as they are small enough to enable fast training, have been thoroughly evaluated in the related work, and many published results are available. Another desirable property is that they all have the same spatial size of 32×32 pixels and can therefore be evaluated using very similar models. Table 6.1 provides an overview of the characteristics of the datasets and figure 6.1 a visualization, respectively.

For the experiments in chapter 7, we sample 10% of the examples contained in each training set randomly, and use it as hold-out validation data.

	dimensionality	**classes**	**training set**	**test set**
MNIST	784 ($1 \times 32 \times 32$ grayscale)	10	60,000	10,000
SVHN	3072 ($3 \times 32 \times 32$ color)	10	73,257	26,032
CIFAR10	3072 ($3 \times 32 \times 32$ color)	10	50,000	10,000
CIFAR100	3072 ($3 \times 32 \times 32$ color)	100	50,000	10,000

Table 6.1: Characteristics of MNIST, SVHN, CIFAR10, and CIFAR100



(a) MNIST (b) SVHN (c) CIFAR10 (d) CIFAR100

Figure 6.1: Sample images from MNIST, SVHN, CIFAR10, and CIFAR100

6.1.1 MNIST

The MNIST database of handwritten digits [39] is a subset of a larger dataset from the National Institute of Standards and Technology (NIST). The dataset contains handwritten digits from zero to nine that have been size-normalized and centered in a fixed size image. As of this writing, the best result achieved on the MNIST dataset is an accuracy of 99.77% with a convolutional neural network using dropconnect [65].

The MNIST dataset is approximately balanced, meaning that there are an almost equal number of examples for each class of the dataset.

6.1.2 SVHN

The SVHN dataset is a real-world dataset of house numbers in Google Street View images. The digits range from zero to nine and have been resized to a fixed resolution of 32×32 pixels, a format that should resemble MNIST. However, the dataset is slightly harder to generalize than MNIST since the images contain not only the digit of the ground truth label, but also some distracting digits of the adjacent numbers. Currently, the best accuracy achieved on SVHN is 98.31% and features a convolutional neural network with mixed, gated, and tree pooling [40].

The SVHN dataset is a highly unbalanced dataset which stems from the fact that lower numbers occur more frequently in real world house numbers. For instance, the most frequently occurring house number is the number one, which is included more than three times as often as the number nine. This property of SVHN may lead to a very biased estimate and similarly high fluctuations of both validation and test accuracy in our experiments. Taking the previous example, by guessing that every number of the dataset belongs to class one, our classifier already achieves a decent accuracy without having learned anything. On the other hand, especially since the number one occurs more frequently in real life situations, the unbalanced frequencies can be seen as prior knowledge of the house number classification task itself.

6.1.3 CIFAR10

The CIFAR10 dataset is a subset of the 80 million tiny images dataset [32]. The dataset contains small images divided into ten classes of objects and animals such as automobiles, trucks, and horses, for instance. The classes are completely mutually exclusive, i.e. there is no overlap between the automobile and truck classes, for example. Currently the best accuracy achieved on CIFAR10 is 96.53% with a convolutional neural network using fractional max-pooling [17].

The CIFAR10 dataset is perfectly balanced, meaning that the frequency of class labels is exactly equal for all classes.

6.1.4 CIFAR100

The CIFAR100 dataset is also a subset of the 80 million tiny images dataset [32]. It also contains small images but they are divided into 100 fine classes of objects, animals, and humans such as bottle, lobster, and girl, respectively. The best accuracy for CIFAR100 is 75.72% and is achieved by a convolutional neural network featuring exponential linear units [5].

The CIFAR100 dataset, just like CIFAR10, is perfectly balanced and contains an equal number of examples for each class.

6.2 Models

All of the models defined below use global feature standardization (section 2.3.1) of the input data and the cross entropy loss (section 2.11.2). We use Nesterov accelerated stochastic gradient descent with a momentum coefficient of $\mu = 0.9$ (section 2.7.1). We acknowledge the superiority of adaptive methods for optimization but will refrain from using them for our experiments since the parameter updates are significantly harder to interpret. We use an adaptive learning rate decay schedule based on the validation accuracy measured after each training epoch (section 2.8.2). The learning rate is decayed by a factor of $k = 0.5$ whenever the validation accuracy shows no improvement for $t = 5$ epochs. The maximum number of epochs without improvements is set to $t_{\max} = 20$. The models do not use any form of weight regularization, in other words, the regularization strength is set to $\lambda = 0$ and the total loss only consists of data loss, and no regularization loss (section 2.12.2). We train the models on all of the datasets in table 6.1 with a batch size of $\mathcal{B} = 64$ for a maximum of 100 epochs, unless otherwise specified.

6.2.1 Multi-layer perceptron

The first model is a naive multi-layer perceptron (referred to as MLP from here on) based on the toy model used in the batch normalization paper [28] with three fully connected layers of 100 activations each. The output of each fully connected layer is followed by a nonlinearity. If batch normalization is added, it is placed between

each linear layer and the nonlinearity. The full model description is presented in table 6.2.

Layer	Description
`input`	32×32 color or grayscale image
`reshape`	flatten
$3\times$ `linear`	100 hidden units each
`softmax`	c-way softmax

Table 6.2: Multi-layer perceptron (MLP) model description

6.2.2 Convolutional neural network

The second model is a deep convolutional neural network based on the VGG network [53] which is referred to as CNN from here on. The model features 10 convolutional layers using 3×3 kernels and both zero-padding and stride set to 1×1 in order to preserve the spatial extent of the input when performing convolutions. The convolutional layers appear in groups of two with the same amount of filters and are followed by max pooling layers with pooling regions of size 2×2 and 2×2 strides. The convolution-pooling layers are followed by two fully connected layers with 512 hidden units. Each convolutional and linear layer is followed by an activation function. If batch normalization is added, it is placed between each convolutional or linear layer and the following nonlinearity. If dropout is added, it is placed in between the convolution operations of each pair of subsequent convolution layers. The final predictions are computed with a c-way softmax, where c denotes the number of classes. The full model description is presented in table 6.3.

This particular model was chosen mainly because of its expressive power and simplicity. We acknowledge that models that use inception blocks [58], residual connections [20] or both [59] perform significantly better in terms of final accuracy but at the cost of reduced interpretability. The reader is referred to the respective literature for discussions of more advanced model architectures.

Layer	Description
`input`	32×32 color or grayscale image
$2\times$ `convolution`	64 filters each
`maxpool`	output size: 16×16
$2\times$ `convolution`	128 filters each
`maxpool`	output size: 8×8
$2\times$ `convolution`	256 filters each
`maxpool`	output size: 4×4
$2\times$ `convolution`	512 filters each
`maxpool`	output size: 2×2
$2\times$ `convolution`	512 filters each
`maxpool`	output size: 1×1
`reshape`	flatten
$2\times$ `linear`	512 hidden units each
`softmax`	c-way softmax

Table 6.3: Convolutional neural network (CNN) model description

Chapter 7

Experimental results

This section presents the experimental results obtained when training both the MLP and CNN model on the four image datasets. All plots in this section contain a subfigure for each dataset the model has been trained on. The *vanilla* network, i.e. the one not using batch normalization, is denoted by the color blue, whereas the *batch normalized* network is denoted by the color green. We generally denote the validation accuracy with solid lines and the cross entropy loss with dashed lines, unless otherwise specified. We stop training the network when we encounter 20 consecutive epochs without validation accuracy improvement in order to speed up the experiments. The upper bound for the number of training epochs is 100 epochs.

The aim of the experiments is to highlight the properties of batch normalization using different datasets, model architectures, and hyperparameters. We do not attempt to match or improve state-of-the-art results on any of the datasets, but to isolate the effects of batch normalization in comparison to vanilla models. Nevertheless, appendix B contains tables with the final test accuracies for both models on all datasets.

In order to make the experiments comparable for all datasets and models, we find it crucial to plot the validation accuracy on absolute scales. The purpose of this is to highlight the increasing difficulty of the datasets, but also the absolute differences in validation accuracy. Conversely, the cross entropy loss is visualized on a relative scale starting at zero up to the largest observed numerical value. Since these values can differ significantly from dataset to dataset, the cross entropy loss does not share a common axis, nor are the actual values shown. As an example, a model trained on a dataset with 10 classes – such as MNIST, SVHN, and CIFAR10 – exhibits initial loss values of around $\mathcal{L} \approx 2.3$, assuming random guessing. For CIFAR100 with 100 different labels, the initial loss value defaults to $\mathcal{L} \approx 4.6$. This behavior follows immediately from the definition of the cross entropy loss function with the natural logarithm.

7.1 Multi-layer perceptron

In this section, we experimentally verify some of the benefits of batch normalization using a small multi-layer perceptron model (MLP). We conjecture that this arguably simple model gives us the ability to reason about the effects of batch normalization on a small scale, such that we can extrapolate our findings to the larger model and gain valuable insights for more advanced model architectures.

7.1.1 Activation functions

In the following sections, we will equip the MLP model with both rectified and saturating activation functions in order to assess the effect of batch normalization when using different nonlinearities. We initialize the ReLU and ELU models according to the Kaiming initialization scheme, whereas the sigmoid model is initialized according to the Xavier initialization scheme. All remaining hyperparameters remain unchanged.

ReLU activation function (baseline model)

For the following experiments, we equip the MLP model with ReLU nonlinearities in order to create a toy model that resembles the current state-of-the-art in activation functions. We will use this model as a baseline for all the following experiments. We initialize the weights according to the Kaiming initialization scheme, which allows for a more stable learning process when using rectifiers. All other hyperparameters remain unchanged. Figure 7.1 shows the cross entropy loss (dashed line) and validation accuracy (solid line) over time for both models trained on all four datasets.

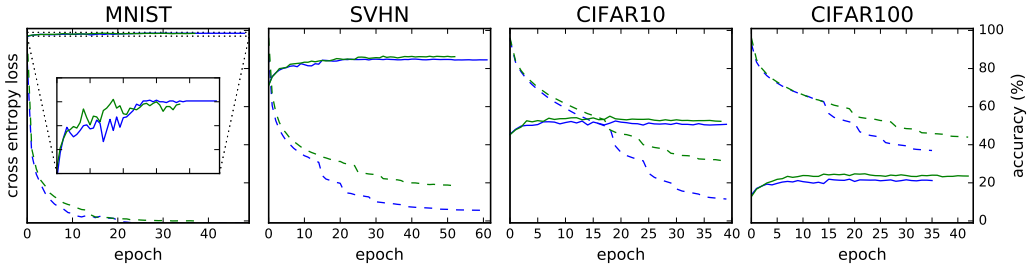


Figure 7.1: Training loss and validation accuracy for MLP with ReLU activations.

The batch normalized ReLU models consistently achieve higher validation accuracies but gain no immediate convergence speed improvements on any of the datasets. In terms of validation accuracy, the gap between the two models is noticeable but not revolutionary.

By observing the loss over time, the regularizing effect of batch normalization becomes very prominent. One can clearly see how the loss repeatedly falls and

plateaus in the vanilla model whenever the learning rate is annealed. The step-wise behavior of the loss is far less significant in the batch normalized model, which consistently achieves higher validation accuracies. Although near-zero losses on the training set are an indicator of overfitting, we cannot observe decreasing validation accuracies over time, whether batch normalization is used or not. A prime suspect for this behavior is the low model complexity, which prevents the model from overfitting the large amount of training examples.

In a second experiment, we added the batch normalization layers after the nonlinearity and achieved the exact same results. The authors of batch normalization strictly recommend that the respective layers should be added before the nonlinearity. We find it important to point out that we cannot see any direct improvements that arise from placing the batch normalization layer before the nonlinearity, at least for our toy model.

Sigmoid activation function

For the following experiments, we use sigmoid nonlinearities after each fully connected layer and initialize the network weights according to the Xavier initialization scheme as discussed in section 2.4.2. We perform this experiment since models with saturating nonlinearities can reportedly be trained with batch normalization, regardless of their well-known inferior properties. Figure 7.2 shows the cross entropy loss (dashed line) and validation accuracy (solid line) over time for both models.

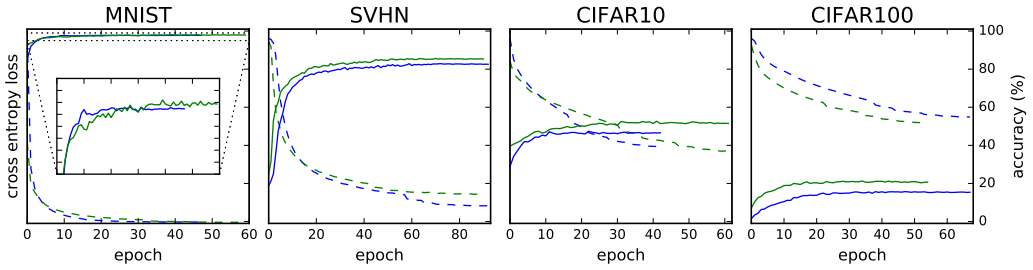


Figure 7.2: Training loss and validation accuracy for MLP with sigmoid activations.

In terms of validation accuracy, we can observe that we reach a higher overall validation accuracies in all cases by simply adding batch normalization before each sigmoid nonlinearity, and leaving all other hyperparameters unchanged. For the digit datasets, batch normalization achieves a validation accuracy that only exceeds the vanilla network by a small margin. For the object datasets, the difference between vanilla and batch normalized network is notably more significant, with validation accuracies deviating by a couple of percent. Furthermore, we can observe that in addition to reaching a higher validation accuracy in all cases, the batch normalized model converges to a beneficial parameter state slightly quicker.

A noteworthy observation can be made in the beginning of training the vanilla network on SVHN. During the first few epochs, the validation accuracy plateaus

for a short amount of time, until the gradient descent updates escape the unfavorable local minimum. A reasonable explanation for this phenomenon is suboptimal initialization, although Xavier initialization was used in this particular case. The batch normalized network does not exhibit this behavior and converges instantly without plateauing.

Regarding the behavior of the cross entropy loss over time, we notice that the slope of the batch normalized loss is much steeper right in the beginning of the training process, after which it decelerates and plateaus slightly quicker than the vanilla network in all cases but CIFAR100. For this particular dataset, we hypothesize that our toy model simply does not have the expressive power to reliably predict 100 classes with three hidden layers of 100 activations each. This can be verified by looking at both the plateauing accuracy and cross entropy loss. Nevertheless, the behavior of the loss is the first hard evidence for both the accelerating and regularizing effects of batch normalization, mainly since we gain higher validation accuracies with simultaneously higher losses.

Thus far, we conclude that batch normalization leads to quicker convergence, higher validation accuracies, and has a regularizing effect, at least for the model used in the experiments. This begs the question if we can use batch normalization in conjunction with ELU nonlinearities and still gain noticeable improvements.

ELU activation function

The aim of this section is to verify the following claims made by [5] about their experiments on the CIFAR100 dataset, not to simply compare different activation functions. To reiterate, they claim that ELU networks outperform ReLU networks with batch normalization and that batch normalization does not improve ELU networks in terms of validation accuracy and convergence speed. Figure 7.3 shows the training loss and validation accuracy for MLP with ELU nonlinearities, trained on all four datasets.

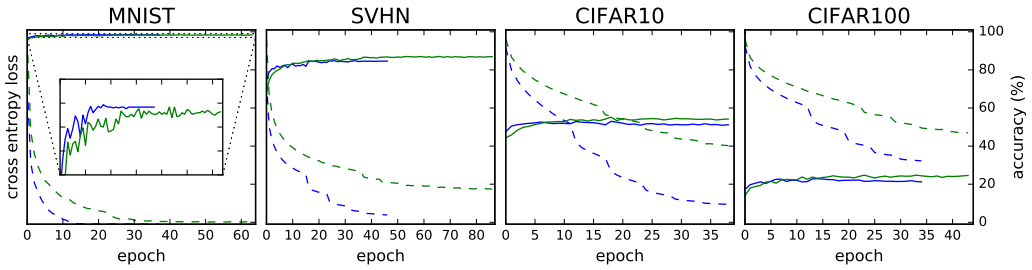


Figure 7.3: Training loss and validation accuracy for MLP with ELU activations.

The first observation is that batch normalized networks seem to converge slower than their vanilla counterparts when using ELU activations, but achieve a higher validation and test accuracy for all datasets except MNIST. As a caveat, the vanilla network beats the batch normalized model by a slight margin on the validation set,

and by an even smaller margin on the test set. We find these differences to be overly small to give a definitive statement about the effectiveness of ELU activations at this point. However, the accelerated convergence of the vanilla model in general and the inferior validation accuracy of the batch normalized model on MNIST are noteworthy.

Comparison of ReLU and ELU activations

Since the network with sigmoid activations performs significantly worse than the models using rectifiers, we will limit our comparison to the models using ReLU and ELU nonlinearities. Figure 7.4 shows the vanilla and batch normalized model's validation accuracy for both ReLU and ELU activation functions.

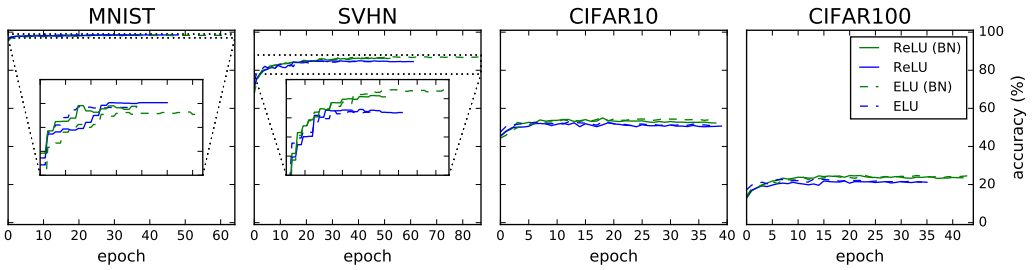


Figure 7.4: Validation accuracy comparison for MLP with ReLU and ELU activations.

Based on our experiments, we cannot confirm the results in [5], and argue that batch normalization does in fact improve ELU networks by a slight margin. Similarly, the batch normalized network with ELU activations outperforms the ReLU network for all datasets but MNIST, if only with a very slight margin. We find these results to be to too insignificant to definitively assess the superiority of ELU activations.

7.1.2 Initial learning rate

The authors of batch normalization argue that the method enables the network to cope with higher learning rates without the risk of divergence and other ill side effects. Therefore, we alter the initial learning rate for the following experiments in oder to assess their claim. The experiments below make use of our baseline model, namely the one using ReLU nonlinearities.

Increased initial learning rate

After observing that batch normalization also has a positive effect on networks with different nonlinearities, we will verify the claim that batch normalized networks can be used with higher initial learning rates without the risk of divergence. We

increase the learning rate by a factor of ten, i.e. set the learning rate to $\eta = 0.1$ for the following experiments. Figure 7.5 shows the cross entropy loss and validation accuracy over time for both models.

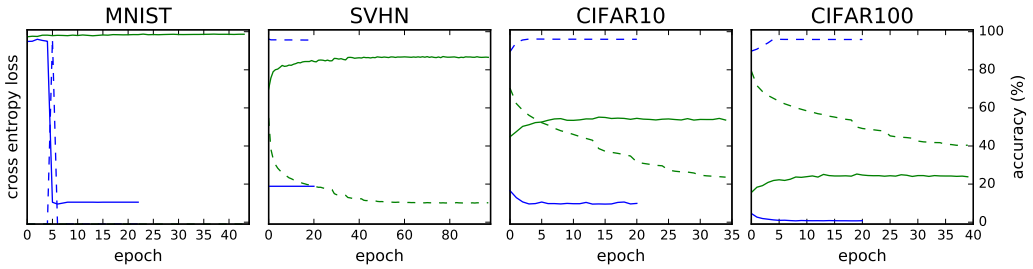


Figure 7.5: Training loss and validation accuracy for MLP with ReLU activations and increased initial learning rate $\eta = 0.1$.

For all datasets, the vanilla model suffers from severe divergence and cannot reach test accuracies that match the ones of the batch normalized model. The batch normalized model, on the other hand, does even better than its counterpart with lower learning rates, consistently achieving higher validation accuracies.

An interesting observation can be made for the vanilla model trained on MNIST. This particular model seems to converge just as the batch normalized model in the beginning of training. Around epoch 5, we see a large spike in the loss and the model quickly forgets everything it has learned up to this point.

After examining the gradients propagated throughout training, we conclude that the training behavior of the vanilla network was caused by exploding gradients. Figure 7.6 shows the average gradients over the course of training for both the vanilla and the batch normalized network. For the vanilla network trained on all datasets, learning essentially stops after the gradients explode right in the beginning of training. This gradient explosion results in many dead neurons caused by the rectified activations. A gradient descent update made almost all neurons output negative values, which cannot be forward propagated and thus have no effect on the error backpropagation.

Since the gradients are proportional to the weights and the network trained on MNIST has lower dimensionality with the Kaiming initialization scheme, the magnitude of the gradients is proportionally lower in this case.

The exploding gradient problem is a phenomenon which could have been avoided by clipping the gradients when their norm is above a certain threshold as in

$$\nabla_{\boldsymbol{\theta}} \mathcal{L} \leftarrow \nabla_{\boldsymbol{\theta}} \mathcal{L} \cdot \frac{\omega}{||\nabla_{\boldsymbol{\theta}} \mathcal{L}||} \tag{7.1}$$

where ω is the threshold and the gradients are only clipped if $||\nabla_{\boldsymbol{\theta}} \mathcal{L}|| > \omega$.

However, keeping the gradients in a certain range of values may restrict the model and ultimately introduces more complexity, which makes an even stronger case for batch normalization. Additionally, gradient clipping introduces yet another

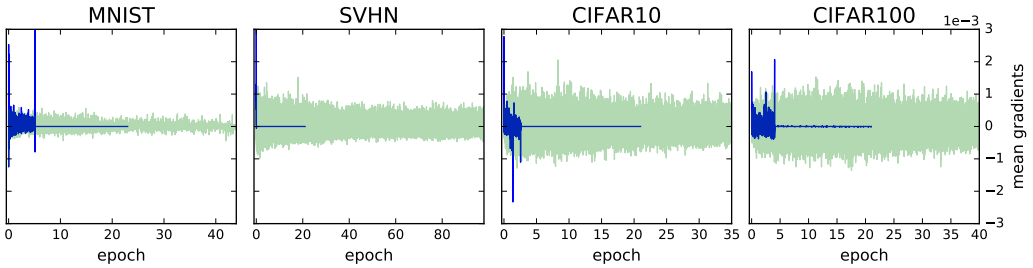


Figure 7.6: Mean gradients for MLP with ReLU activations and increased initial learning rate $\eta = 0.1$.

hyperparameter that is not trivial to choose correctly without observing the model's divergence beforehand. For a rigorous analysis of the exploding gradient problem and its avoidance, the reader is referred to [45].

Decreased initial learning rate

Now we perform the same experiment with the learning rate set to $\eta = 0.001$, which is ten times less than the baseline learning rate of $\eta = 0.01$.

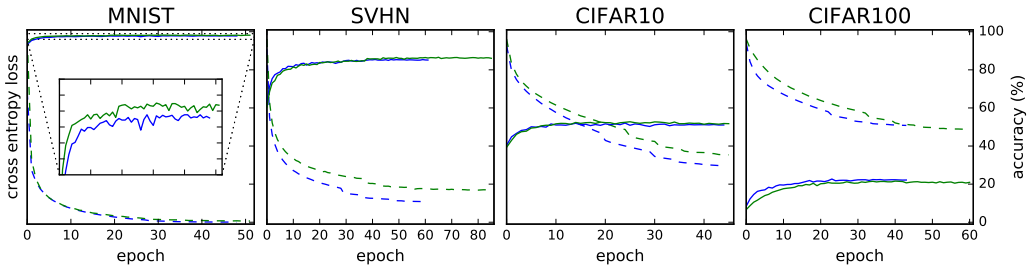


Figure 7.7: Training loss and validation accuracy for MLP with ReLU activations and increased initial learning rate $\eta = 0.001$.

Interestingly, decreasing the learning rate diminishes the effect of batch normalization almost entirely. For all datasets, the batch normalized network overfits slightly, as does the vanilla network. In case of CIFAR100, the batch normalized network even performs worse than its vanilla counterpart. Apart from generally higher loss values of the batch normalized network, the models behave very similarly.

Comparison of initial learning rates

Since the vanilla network shows severe divergence for increased learning rates, we will limit our comparison to the batch normalized models. Figure 7.8 shows the validation accuracies for the batch normalized models trained on all datasets with different initial learning rates.

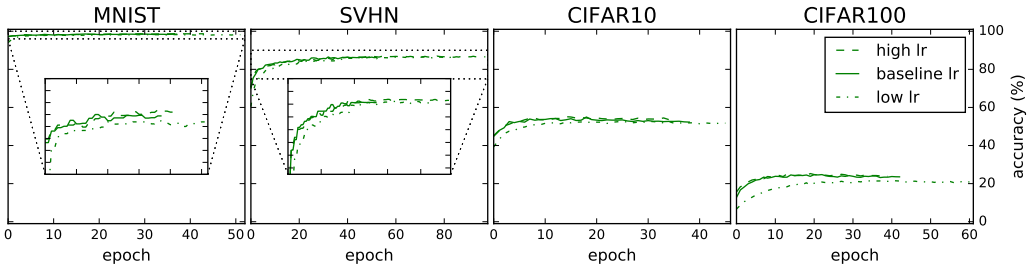


Figure 7.8: Validation accuracy comparison for MLP with ReLU activations and different learning rates.

From the direct comparison, we can observe that the models with a decreased initial learning rate perform strictly worse that the baseline and high learning rate models, both in terms of validation accuracy and convergence speed. Interestingly, the batch normalized model is in fact able to train with significantly higher learning rates, and still outperform the lower learning rates without causing divergence. As a consequence of this behavior, we recommend to experiment with increased rather than decreased learning rates when batch normalization is employed.

7.1.3 Weight initialization

So far, we have been using the theoretically optimal initialization scheme for the given activation function, i.e. Xavier initialization for saturating nonlinearities and Kaiming initialization for rectified nonlinearities. In general, it is not advised to use saturating nonlinearities but we will include the experiments for the sake of completeness. One of the alleged effects of batch normalization is that one needs to be less careful about weight initialization in general. To verify this allegation, we will purposefully initialize the weights to non sensical values, as to find out how it will affect the network behavior.

High variance weight initialization

As in the previous experiments, we only change one hyperparameter, namely the weight initialization scheme and leave all other hyperparameters constant. We sample the weights from a normal distribution with mean of zero and standard deviation of one. The experimental results can be obtained in figure 7.9.

During all experiments with the vanilla network, the cross entropy loss gets unstable right in the beginning of training and its values explode, which is the reason for the missing training loss in the figures. For all datasets, the network is not able to recover from the exploding loss values and thus never reaches higher accuracies than random guessing. For MNIST, CIFAR10, and CIFAR100, the validation accuracy settles around the expected value for random guessing, given the number of classes.

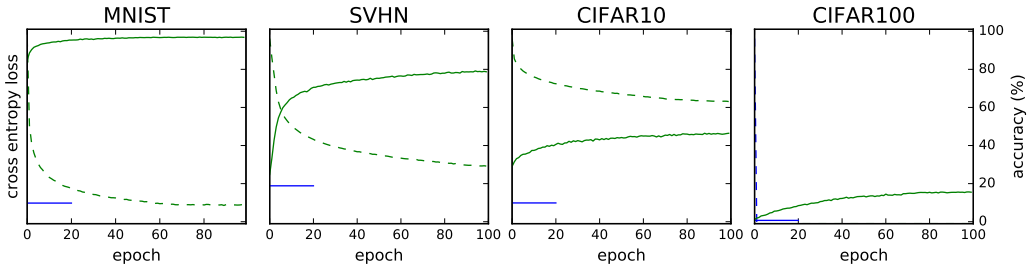


Figure 7.9: Training loss and validation accuracy for MLP with ReLU activations and weight initialization with standard deviation $\sigma = 1$.

Since SVHN is a highly unbalanced dataset, we see much better accuracies for random guessing.

The batch normalized network does not exhibit exploding loss values and is able to converge in a normal fashion, although the convergence is a lot slower than with proper initialization schemes such as Kaiming. Both the convergence speed and the final test accuracies are strictly worse than their properly initialized counterparts.

We can conclude that batch normalization does in fact let us be less careful about weight initialization since the network is strictly able to reach favorable parameter states, which is not the case for the vanilla network.

Low variance weight initialization

In the following experiments, we initialize the weights with small gaussian values with mean zero and standard deviation of $\sigma = 0.0001$. The experimental results can be obtained in figure 7.10.

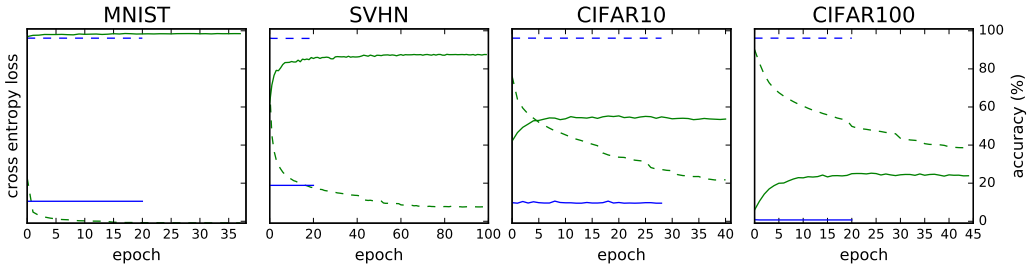


Figure 7.10: Training loss and validation accuracy for MLP with ReLU activations and weight initialization with standard deviation $\sigma = 0.0001$.

This experiment clearly verifies the theoretical result that smaller weights lead to larger gradients in batch normalized networks, which is the reason for its increased convergence speed.

Similar to the high variance initialization of the weights, the vanilla network is not able to reach a favorable parameter state, and thus never achieves higher

accuracies than random guessing would.

Comparison of weight initializations

Since the vanilla models are not able to reach favorable parameter states in any case, we will restrict our comparison to the batch normalized models. Figure 7.11 visualizes our comparison.

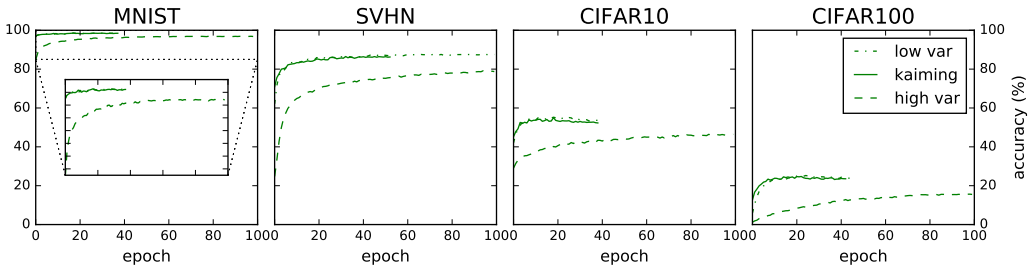


Figure 7.11: Validation accuracy comparison for MLP with ReLU activations and different weight initializations.

It is immediate from this visualization that high variance weight initialization performs strictly worse than the baseline or low variance initialization. Again, this result is in accordance to the properties of batch normalization, namely that smaller weights lead to larger gradients and vice versa. Interestingly, the low variance initialization actually seems to improve batch normalized models by a very slight margin. This result calls for further investigation whether increasingly smaller weights lead to favorable convergence properties. The magnitude of the gradients in case of high variance weight initialization are simply not sufficient to reach favorable parameter states fast enough. In our experiments, we stopped training after 100 epochs, which does not rule out the possibility of the high variance network eventually reaching competitive performance in terms of validation accuracy.

7.1.4 Regularization

As discussed before, regularization is an important technique to improve generalization performance by reducing overfitting in neural networks and statistical models in general. Although the MLP model does not suffer from overfitting but rather underfits the data due to its simplicity, we find it important to analyze the convergence behavior when regularization techniques are added to the model.

Dropout

Figure 7.12 contains the training loss and validation accuracy for the MLP model trained on all four datasets with dropout and keep probability of $p = 0.5$.

Although the use of dropout does not improve our baseline model on any dataset, the batch normalized model outperforms the vanilla model on all datasets in terms

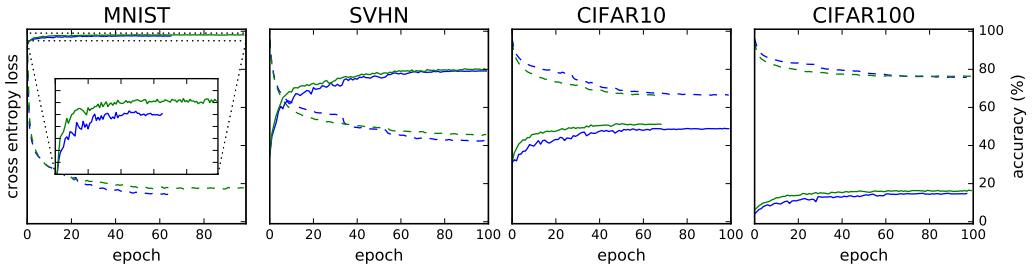


Figure 7.12: Training loss and validation accuracy for MLP with ReLU activations and dropout with keep probability $p = 0.5$.

of convergence speed and accuracy. A noteworthy observation is that – although the learning rate decays repeatedly – the step-wise behavior of the loss is now way less severe than in our baseline model. This adds to the evidence that batch normalization does in fact have some of the same properties as dropout in terms of its regularizing effect.

Generally speaking, the loss does not go down nearly as much as without dropout. Again, we conjecture that the three hidden layer model has nowhere near the representational power to classify any of the non-digit datasets accurately, especially if half of the units are dropped during training.

Weight decay

Figure 7.13 visualizes the training loss and validation accuracy for the MLP model trained on all four datasets with L_2 regularization strength of $\lambda = 0.0005$.

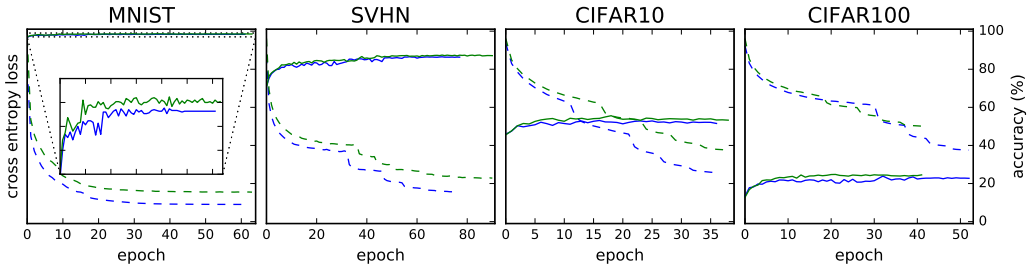


Figure 7.13: Training loss and validation accuracy for MLP with ReLU activations and L_2 weight decay with regularization strength $\lambda = 0.0005$.

Adding weight decay achieves higher accuracies than our baseline model, whereas the relative effect of batch normalization on the convergence speed and accuracies diminishes slightly. This corresponds to the recommendation of the authors in [28] who argue that L_2 weight regularization can be reduced when batch normalization is employed.

As can be seen from this experiment, adding L_2 weight regularization has a strictly positive effect on the generalization performance, and we conclude it can safely be used in conjunction with batch normalization. Additionally, we can verify that batch normalization regularized the model further by observing that we achieve higher accuracies with simultaneously higher losses.

Comparison of regularization methods

As before, the batch normalized models strictly outperform the vanilla models, which is why we restrict our comparison to those models. Figure 7.14 visualizes our comparison.

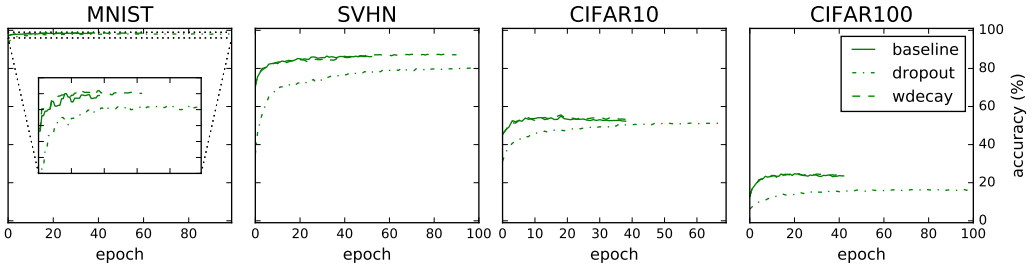


Figure 7.14: Validation accuracy comparison for MLP with ReLU activations and different regularization methods.

It is clear from the visualization that dropout does not improve our batch normalized models. However, we attribute this behavior to the fact that the models did not need regularization in the first place because of their limited representational power. More importantly, the dropout keep probability of $p = 0.5$ is too high in most cases since we are restricting our already limited model to only perform updates with half its activations.

A small amount of weight decay seems to improve our generalization performance by a very slight, but noticeable margin. We conclude that weight decay can be used in conjunction with batch normalization without ill side effects.

7.1.5 Batch size

For the following experiments, we alter only the batch size used for training, and leave all other hyperparameters constant. We select the batch sizes in powers of two as in $\mathcal{B} = \{2^5..., 2^{10}\}$. These batch sizes are chosen because we see an exponential rise in training time for batch sizes $\mathcal{B} < 2^5$ and a batch size of $\mathcal{B} > 2^{10}$ reaches the memory limit of our GPU for the CNN model. To ensure comparable experiments, we select the hyperparameters in exactly the same way for both models.

Altering the batch size should not have a huge effect on the vanilla network theoretically, but we expect slightly different behavior for the batch normalized network since the batch statistics depend strongly on the size of the batches. We

train the MLP network on all datasets for one hundred epochs, which means the networks with smaller batch sizes will have gone through a lot more individual parameter updates per epoch, since the number of updates is inversely proportional to the batch size. Since small batch sizes do not exploit the hardware parallelization as well, we also see an inversely proportional relationship between wall time and the respective batch size.

Figure 7.15 visualizes the final test accuracy of the MLP network with different batch sizes. Interestingly, the batch normalized network strictly converges to a higher test accuracy for batch sizes smaller than $\mathcal{B} < 2^8$. Likewise, the test accuracy of the batch normalized network decreases monotonically with increasing batch sizes. In case of the vanilla network, we can observe worse test accuracies for networks which have been trained for longer. An inspection of the loss over time verifies that the vanilla networks overfit significantly when trained for too long. Since we perform the evaluation on the test set after 20 consecutive training epochs without improvement, we give the vanilla models a lot of time to overfit and we observe decreased accuracies for smaller batch sizes in this case.

A possible interpretation for this behavior from a regularization perspective is that increasing the batch size makes the batch statistics more washed out, resulting in less accurate parameter updates. Conversely, for smaller batch sizes, the network seems to get just the right amount of regularization, which results in optimal accuracies for the representative power of the toy model.

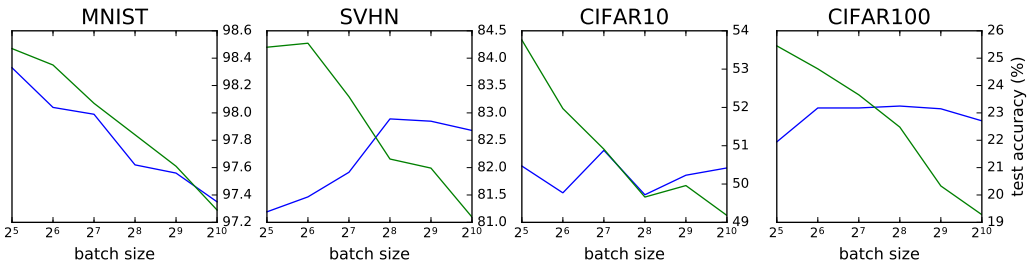


Figure 7.15: Test accuracies for MLP with ReLU activations using different batch sizes.

7.1.6 Epochs vs. wall time

A rather interesting metric when talking about the acceleration of neural network training is to look at wall time rather than the number of epochs, which the authors of batch normalization fail to provide [28]. Adding batch normalization to a network introduces a certain amount of overhead that mostly stems from averaging activations over the entire mini-batch. We acknowledge that it is hard to provide an unbiased measure of wall time since it is heavily dependent on the implementation and the underlying hardware, as well as the system workload during training in general. Over the course of our experiments, we are logging various statistics

of layer activations, weights, and gradients, which introduces a noticeable amount of overhead. However, since the statistics are calculated for the vanilla and batch normalized models alike, we verified the relative time differences to be constant.

For our baseline batch size of $\mathcal{B} = 64$, the wall time per epoch for the vanilla model ranges from 1.8 to 2.7 seconds and from 2.3 to 3.4 seconds for the batch normalized model. Thus, the mean overhead for batch normalization is 27%, or 0.6 seconds per epoch. For a breakdown of wall time per epoch for different batch sizes, the reader is referred to the corresponding table B.3 in the appendix. In general, the overhead induced by batch normalization increases with an increasing number of examples per batch, which follows the intuition that averaging over more examples takes more time.

To put this result into perspective, the average time spent per epoch by the batch normalized model in each plot is 27% higher than the average time spent by the vanilla model. This overhead diminishes the slightly higher convergence speed of batch normalization almost entirely, whereas it clearly has no effect on the validation or test accuracies.

7.2 Convolutional neural network

In this section, we follow the same experimental procedure as before, but replace the MLP model with the CNN model.

As discussed previously, batch normalization is applied to convolutional layers in a different way than for fully connected layers. In the fully connected case, we normalize each activation over the entire mini-batch and learn the batch normalization scale and shift parameters per activation, whereas in the convolutional case, we normalize each feature map over the current mini-batch and learn the scale and shift parameters per feature map, rather than per activation.

7.2.1 Activation functions

We systematically train the vanilla and batch normalized CNN model on all four datasets, and only exchange the activation functions that follow the convolutional of fully connected layers.

ReLU activation function (baseline model)

For the following experiments, we equip the CNN model with ReLU nonlinearities and use Kaiming's initialization scheme as we did for the MLP model. Figure 7.16 shows the training loss and validation accuracy using ReLU activations over time.

Similar to the observations from the toy model, the batch normalized network is slightly ahead of the vanilla one in terms of validation and test accuracy. In case of CIFAR100, the batch normalized network outperforms its vanilla counterpart by a large margin in terms of validation accuracy. Furthermore, we can observe a much higher convergence speed for this dataset.

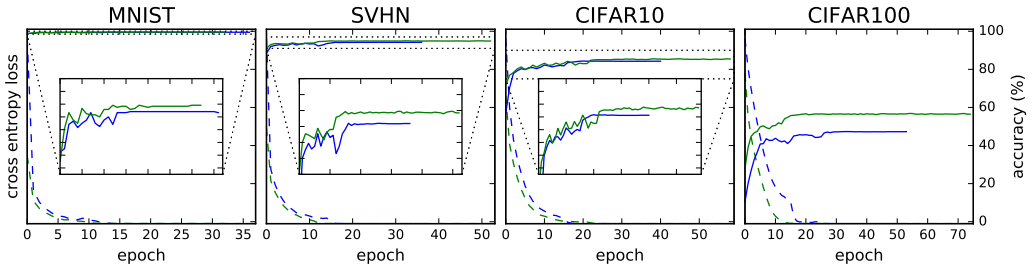


Figure 7.16: Training loss and validation accuracy for CNN with ReLU activations.

Another crucial observation is that the model now has the representational power to overfit to the training set since the loss quickly reaches values around zero and the accuracy on the training set is strictly 100% after only a few epochs. One may argue that we chose a rather complex model for the tasks, especially for MNIST, but with the right type and amount of regularization the model should perform just as well. The zero loss and full training accuracy is true for both the vanilla and the batch normalized network. Whatever regularizing effect batch normalization might have, it is not strong enough to prevent the large model from overfitting to the training set.

Looking at the average gradients propagated through the network, we can observe that the gradients in the vanilla network are generally higher during beginning of training and gradually decrease. Figure 7.17 visualizes this phenomenon.

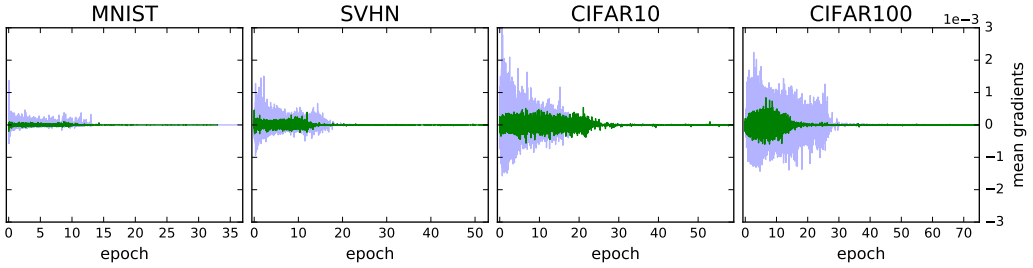


Figure 7.17: Mean gradients for CNN with ReLU activations.

This behavior makes intuitive sense because the parameter updates have far higher magnitudes in the beginning of training since the parameters have to be adjusted more aggressively. In the batch normalized case, however, the gradients magnitude seems to deviate less and stays fairly consistent over time until near zero loss is reached. Even after the loss plateaus close to zero, the batch normalized networks propagates small gradients backwards, whereas the vanilla network's gradients vanish.

Sigmoid activation function

Figure 7.18 shows the training loss and validation accuracy for the CNN model with sigmoid activations.

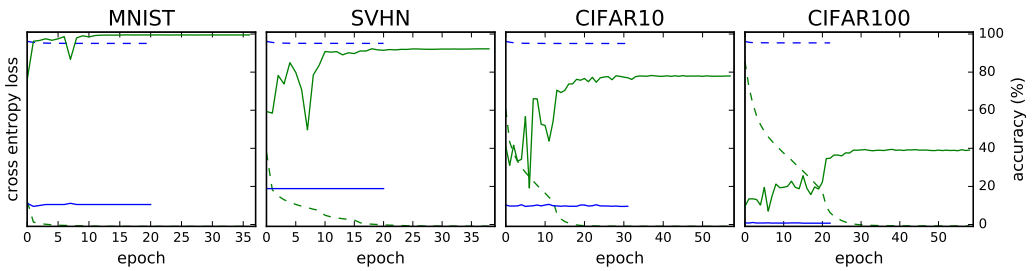


Figure 7.18: Training loss and validation accuracy for CNN with sigmoid activations.

Unlike the MLP model with sigmoid nonlinearities, we can observe that the vanilla network is not able to achieve validation accuracies that outperform random guessing whereas the batch normalized network learns fairly consistently.

After observing the outputs of the individual layers, it turns out that most of the vanilla network's activations were in the saturated regime of the sigmoid nonlinearity, which is a prime example of the vanishing gradient problem. Since the slope of the sigmoid function approaches zero as the absolute value of the activations become larger, the error gradient propagated backwards is also close to zero, i.e. the network does not adjust its parameters. This phenomenon is especially severe as the network depth increases.

The learnable scale and shift parameters of the batch normalization procedure help the network to push its activations closer to the non-saturated regime of the sigmoid activation function and thus enable learning. However, there are severe fluctuations in the validation accuracy over time, which normally suggests that the learning rate was chosen too high and caused divergence. However, when using sigmoid nonlinearities, the network may slowly recover from being stuck in the saturated regime, which we believe is the cause for the validation accuracy fluctuations. Without the scale and shift parameters of the batch normalization procedure, the network is not able to shift the activation values away from the saturated regime fast enough. In order to visualize the vanishing gradient problem, figure 7.19 provides the average gradients flowing back throughout the network over time.

As is obvious from the figure, there are barely any gradients flowing backwards in case of the vanilla network. In other words, the gradients are not large enough to enable jumps to more favorable local minima. The batch normalized model, on the other hand, is able to propagate gradients more effectively

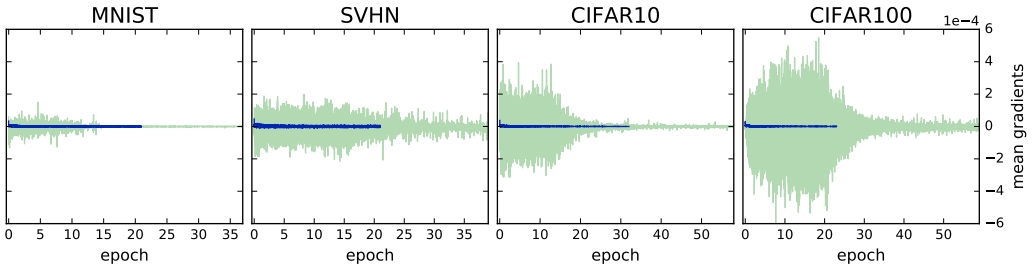


Figure 7.19: Mean gradients for CNN with sigmoid activations.

ELU activation function

Similar to the MLP experiments, we exchange all ReLU nonlinearities with ELU activation functions in order to assess if the batch normalized network performs better than its vanilla counterpart. Again, figure 7.20 shows the training loss and validation accuracy for all four datasets over time.

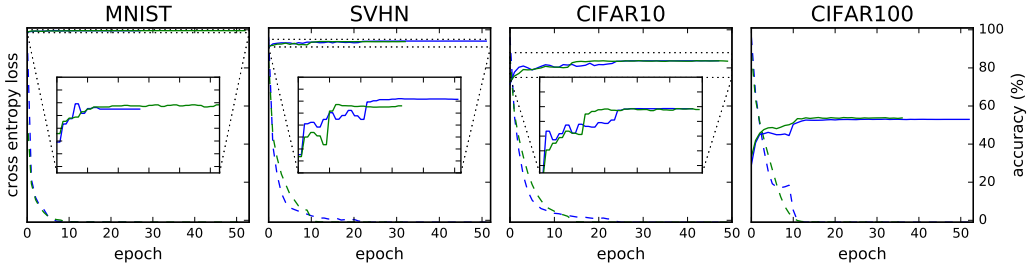


Figure 7.20: Training loss and validation accuracy for CNN with ELU activations.

As can be seen in the figures, the batch normalized network seems to show less improvements when combined with ELU nonlinearities. In fact, neither the convergence speed, nor the final validation and test accuracies are significantly higher than what the vanilla model is able to achieve.

The gradient propagation for the ELU network is slightly better behaved than with the ReLU activations. Figure 7.21 shows the mean gradients propagated over time.

Comparison of ReLU and ELU activations

Since the network with sigmoid activations performs strictly worse than the rectified activations, we will limit the following comparison to ReLU and ELU nonlinearities. Figure 7.22 shows the validation accuracy of the batch normalized network when using ReLU and ELU nonlinearities.

As we can see, the batch normalized network with ReLU activation functions strictly performs best, which is why we take it as a baseline for further experiments.

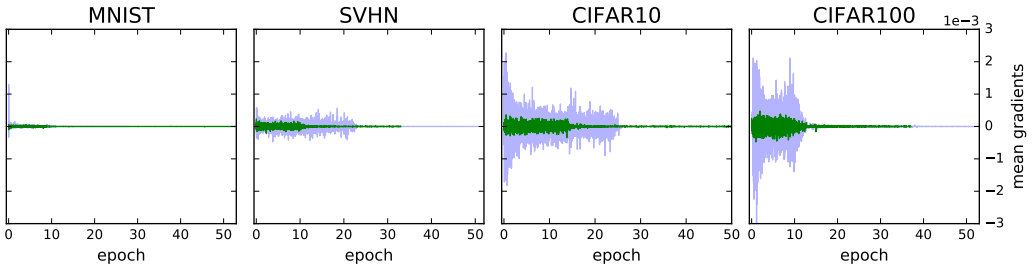


Figure 7.21: Mean gradients for CNN with ELU activations.

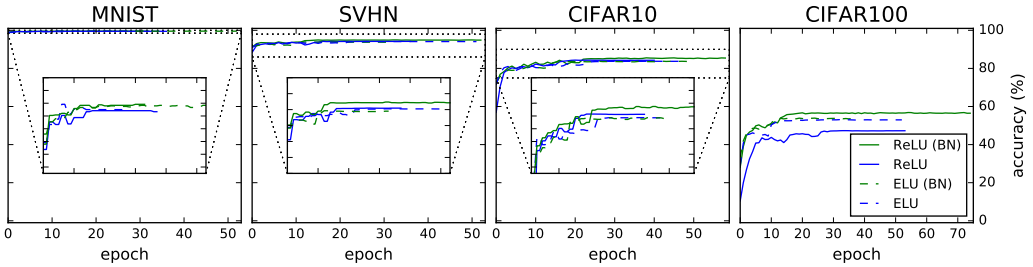


Figure 7.22: Validation accuracy comparison for CNN with ReLU and ELU activations. Solid lines for ReLU and dashed lines for ELU nonlinearities.

We cannot verify the claims made by [5] that ELU networks outperform ReLU networks with batch normalization and that batch normalization does not improve ELU networks in term of validation accuracy and convergence speed. One has to note however, that there is not enough evidence to fully debunk the validity of the second claim.

7.2.2 Initial learning rate

Similar to the MLP experiments, we systematically alter the learning rate of the CNN model to observe the effects of this specific hyperparameter on the performance of both the vanilla and batch normalized network.

Increased initial learning rate

Figure 7.23 visualizes both the training loss and validation accuracy of the CNN model with increased learning rate.

Again, we have exploding gradients in the beginning of training for the vanilla model, after which learning abruptly stops. The batch normalized model does not exhibit exploding gradient behavior and learns almost as if the learning rate had not been changed.

The exploding gradients of the vanilla network are visualized in figure 7.24. In case of CIFAR100, the average gradient is not exploding, but the network is not

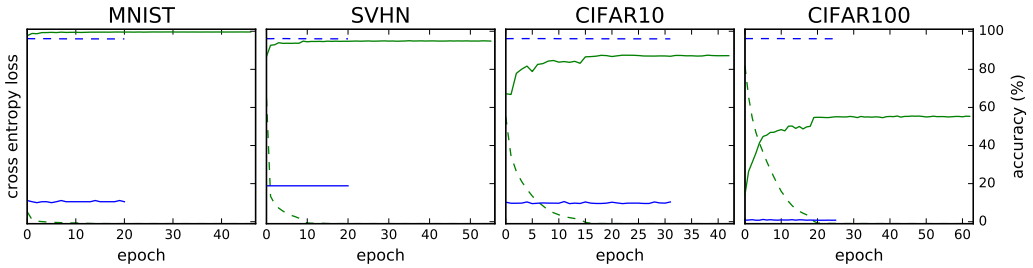


Figure 7.23: Training loss and validation accuracy for CNN with ReLU activations and increased initial learning rate $\eta = 0.1$.

able to learn meaningful, discriminative representations of the images when batch normalization is not used.

In general, the exploding gradient problems seem to persist in almost exactly the same fashion with the CNN model. It has to be noted that the final validation accuracy plateaus very quickly, although we are using an advanced model with more than ten convolutional layers.

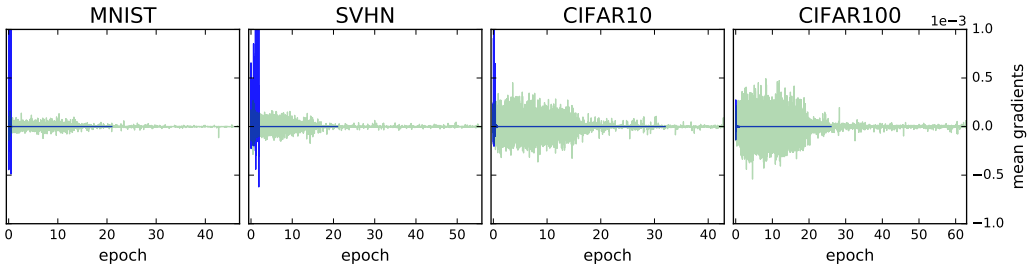


Figure 7.24: Mean gradients for CNN with ReLU activations and increased initial learning rate $\eta = 0.1$.

Decreased initial learning rate

Figure 7.25 shows the training loss and validation accuracy for the CNN model with a decreased initial learning rate.

With the lower initial learning rate, the batch normalized model is further ahead of the vanilla network in terms of convergence speed and validation accuracy, but cannot reach the same accuracies as with the regular or increased learning rate.

The primary observation is that decreasing the learning rate does not seem to have an adverse affect on the batch normalized model, whereas the vanilla suffers from far slower convergence. We conclude that batch normalization also helps the convolutional network converge when the learning rate is too low, which has not been the case for the multi-layer perceptron model.

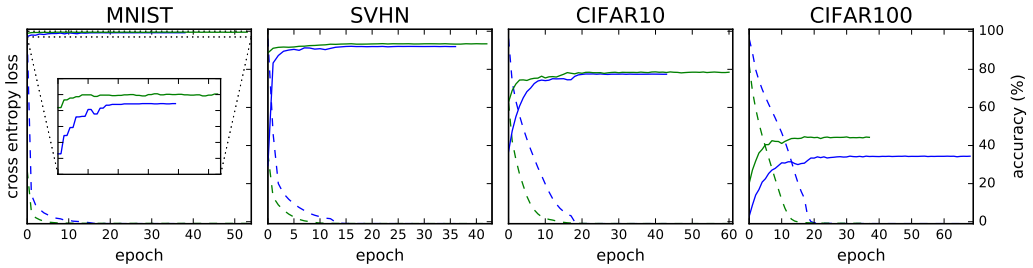


Figure 7.25: Training loss and validation accuracy for CNN with ReLU activations and decreased initial learning rate $\eta = 0.001$.

In case of MNIST and SVHN, the gradient propagation in the vanilla and batch normalized case are fairly comparable. For the CIFAR datasets, however, the gradient magnitude is significantly larger and worse behaved in case of the vanilla model. Figure 7.26 visualizes these phenomena.

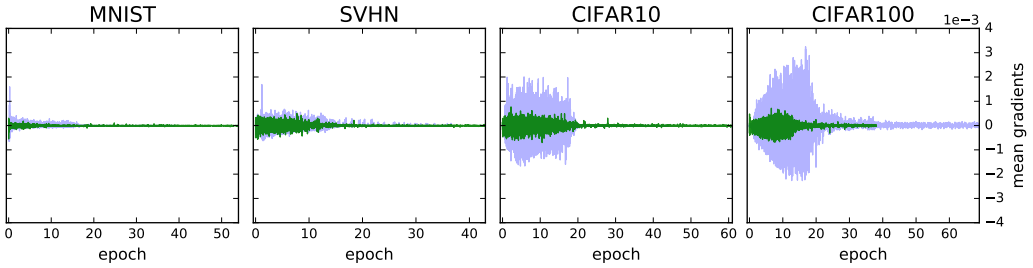


Figure 7.26: Mean gradients for CNN with ReLU activations and decreased initial learning rate $\eta = 0.001$.

Comparison of initial learning rates

Since the batch normalized model strictly performs better than its vanilla counterpart, we will only compare different learning rates for this model. Figure 7.27 shows the validation accuracy of the model with three different initial learning rates.

In all cases but CIFAR100, the increased initial learning rate has a beneficial – or at least no negative – effect on the final validation accuracy of the batch normalized model. As expected, the decreased learning rate has an adverse effect on both the validation accuracy and convergence speed in all cases.

7.2.3 Weight initialization

Similar to the experiments for the MLP model, we systematically alter the weight initialization scheme for the CNN model and train on all four datasets.

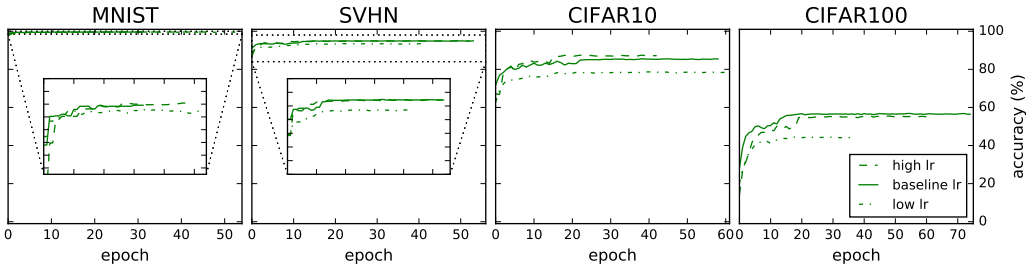


Figure 7.27: Validation accuracy comparison for CNN with ReLU activations and different initial learning rates.

High variance weight initialization

Figure 7.28 visualizes the training loss and validation accuracy for the CNN model with high variance weight initialization.

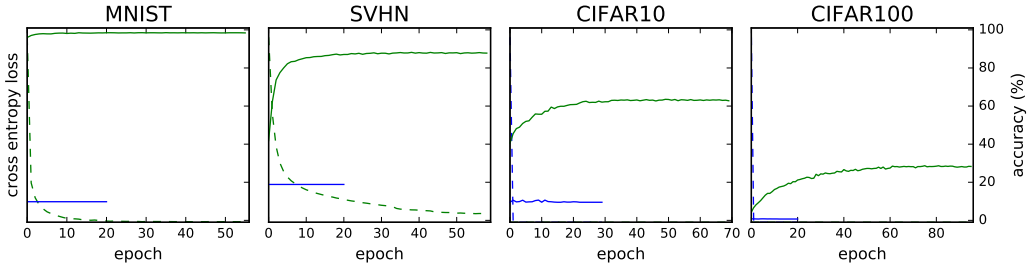


Figure 7.28: Training loss and validation accuracy for CNN with ReLU activations and weight initialization with standard deviation $\sigma = 1$.

The vanilla network cannot achieve validation accuracies better than random guessing. The batch normalized model, on the other hand, learns fairly consistently and is able to achieve favorable parameter states. One has to note, however, that both convergence speed and final accuracies do not nearly match the networks with proper initialization and non-saturating nonlinearities.

As suggested by the stagnating accuracies and losses, the vanilla networks suffer from parameters not updating correctly. An observation of the gradients propagated through the model verifies this speculation. Again, figure 7.29 visualizes the gradients propagated over time.

In case of MNIST and SVHN with the batch normalized model, the gradients are generally higher in the beginning of training, which makes intuitive sense following the earlier discussion. In case of the two CIFAR datasets, the gradient magnitude remains consistent over time, with a slight increase towards parameter updates in the end of training.

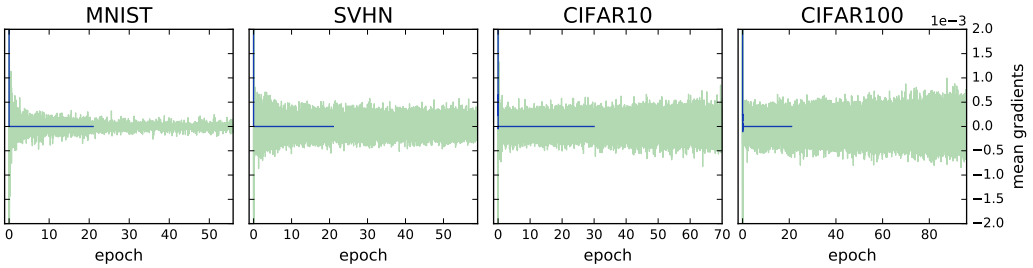


Figure 7.29: Mean gradients for CNN with ReLU activations and weight initialization with standard deviation $\sigma = 1$.

Low variance weight initialization

Figure 7.30 visualizes the training loss and validation accuracy for the models using low variance weight initialization.

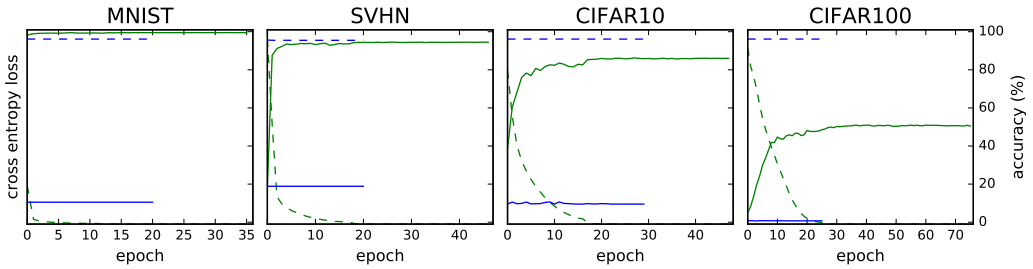


Figure 7.30: Training loss and validation accuracy for CNN with ReLU activations and weight initialization with standard deviation $\sigma = 0.0001$.

Similar observations can be made for the low variance weight initialization model in terms of gradient propagation. However, near-zero losses are reached a lot quicker. This stems from the theoretical results that smaller weights lead to larger gradients in case of using batch normalization. Figure 7.31 visualizes this result.

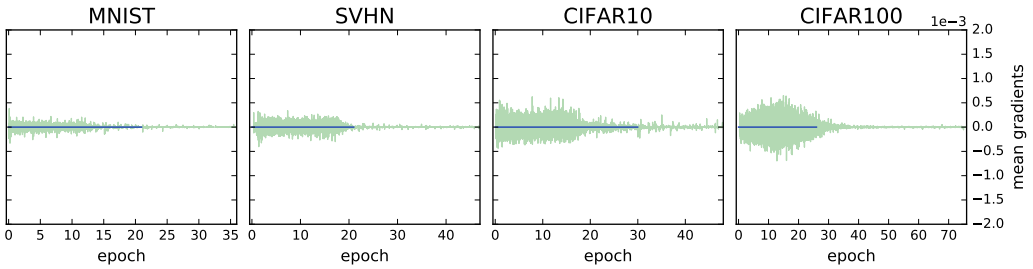


Figure 7.31: Mean gradients for CNN with ReLU activations and weight initialization with standard deviation $\sigma = 0.0001$.

Comparison of weight initializations

Again, the batch normalized network outperforms the vanilla network in all cases, which is why we limit our comparison to that model. Figure 7.32 shows the validation accuracy for all weight initialization schemes discussed above.

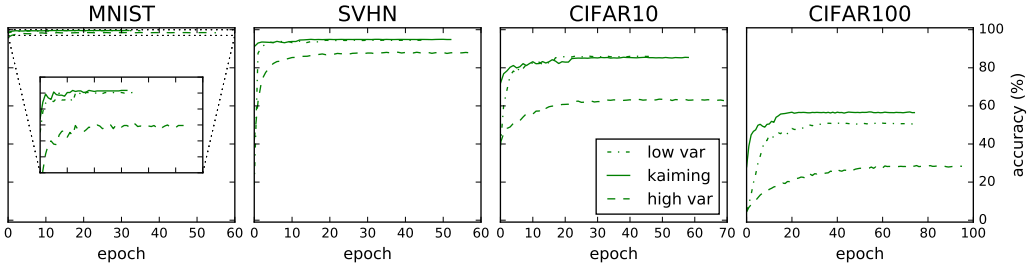


Figure 7.32: Validation accuracy comparison for CNN with ReLU activations and different weight initialization schemes.

As expected, the Kaiming initialization scheme which is especially tailored for rectified nonlinearities beats the suboptimal initialization schemes. Interestingly, the model initialized with low variance Gaussian values performs significantly better than the model with high variance initialization. Again, this behavior is due to the inversely proportional relationship of the gradients and the weights when using batch normalization. The magnitude of the gradients for the high variance model is simply not large enough to perform updates that reach a beneficial parameter state.

Although batch normalization alleviates some of the effects of improper initialization, we can observe that the Kaiming initialization scheme generally produces the best results, both in terms of validation accuracy and convergence speed.

7.2.4 Regularization

In this section, we will focus on some of the most common regularization techniques for state-of-the-art convolutional neural networks as described in section 2.12. The authors of [28] specifically argue that batch normalization reduces the need for dropout and L_2 weight regularization. In order to assess the validity of their statements, we will add both dropout and L_2 weight regularization to the CNN model.

Dropout

Dropout has quickly become one of the most promising techniques for regularization of deep convolutional architectures. It prevents the co-adaptation of features by randomly dropping activations during training as described in section 2.12.3.

We add dropout to the CNN architecture by dropping half the activations between consecutive convolution layers. Similarly, we add a dropout layer between the

last fully connected layer and the softmax layer. We use a dropout keep probability hyperparameter of $p = 0.5$, i.e. we randomly disregard half the activations during training. Figure 7.33 visualized the training loss and validation accuracy for the CNN model with dropout over time.

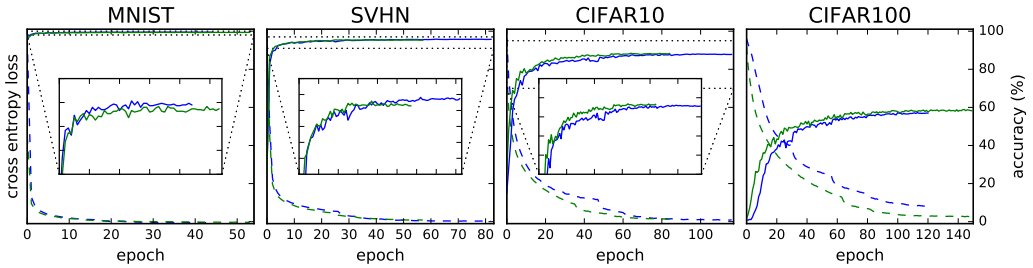


Figure 7.33: Training loss and validation accuracy for CNN with ReLU activations and dropout with keep probability $p = 0.5$.

Generally, the training progresses a lot slower after adding dropout to the model architecture. The vanilla model seems to achieve both higher validation and test accuracies in all cases but CIFAR100. We conclude that batch normalization inhibits some of the effects of batch normalization and therefore concur with the results in [28] that dropout should be removed when using batch normalization. We conjecture that dropout counteracts the effect of batch normalization because the batch statistics do not account for the drop in activations. We have to note, however, that the vanilla models with dropout reach some of the highest validation and test accuracies of all our experiments.

In the following gradient visualization in figure 7.34, we can observe that dropout does in fact have a similar effect as batch normalization, i.e. the gradients are more stable during training. However, we can observe a spiking gradient behavior right in the beginning of training.

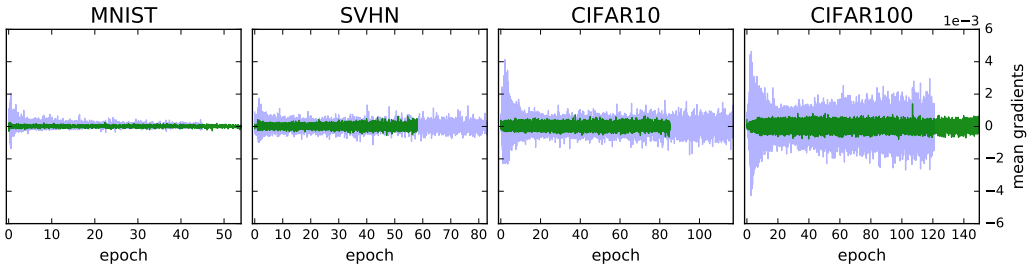


Figure 7.34: Mean gradients for CNN with ReLU activations and dropout with keep probability $p = 0.5$.

Weight decay

So far, the model parameters were unbounded and there growth was only dependent on the current learning rate in conjunction with the backpropagated gradients. By adding a L_2 weight decay regularization term to the cross entropy loss function, we impose a Gaussian prior on the weights of the network as described in section 2.12.2. In other words, we prefer smaller, diffuse weights that have minimal L_2 norm in addition to the data loss computed by the cross entropy loss function. We use a regularization strength hyperparameter $\lambda = 0.0005$ for the following experiments.

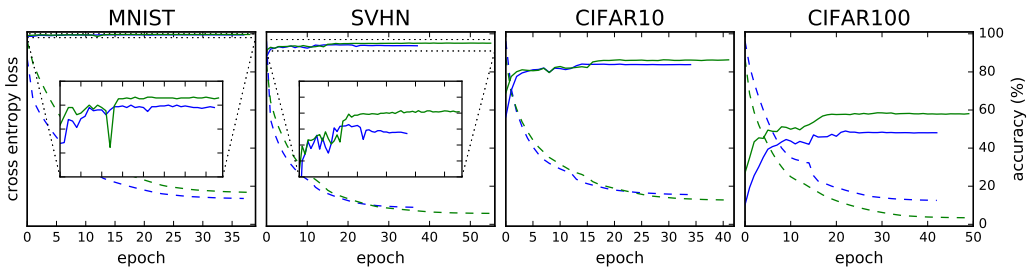


Figure 7.35: Training loss and validation accuracy for CNN with ReLU activations and weight decay with regularization strength $\lambda = 0.0005$.

The first observation is that the loss never reaches values close to zero since the L_2 norm of the weights is added to the loss function. Therefore, for the loss to reach absolute zero, all weights would have to be zero to create a zero L_2 norm which would clearly not lead to meaningful feature extraction.

In case of all four datasets, the batch normalized network beats the vanilla network both in terms of convergence speed and validation accuracy. In case of CIFAR100, the difference is most severe, with almost ten percent improvement for the batch normalized network.

As a conclusion, adding weight decay to the loss function clearly benefits both the vanilla and batch normalized network to gain convergence speed and accuracy improvements.

Observing the gradients propagated throughout training, we find that they do not differ substantially from the baseline model without weight regularization, at least in case of the batch normalized model. The vanilla model's gradients are slightly better behaved, with less fluctuations. The mean gradients of the vanilla and batch normalization models over the course of training are visualized in figure 7.36.

Comparison of regularization techniques

Again, the batch normalized network is ahead of the vanilla model in almost all cases, which is why we limit our comparison to this model. Figure 7.37 shows

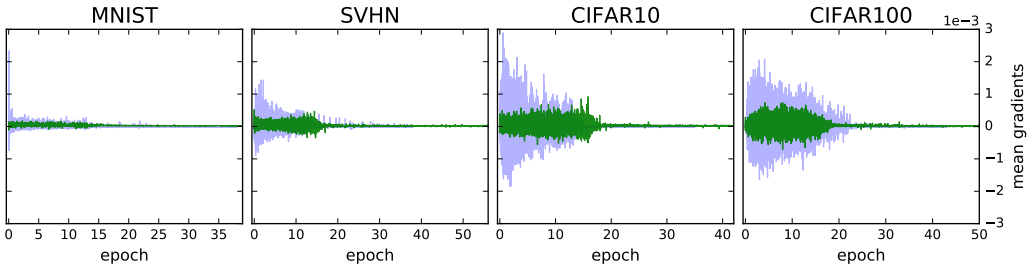


Figure 7.36: Mean gradients for CNN with ReLU activations and weight decay with regularization strength $\lambda = 0.0005$.

the validation accuracy of the batch normalized model using the regularization techniques discussed above.

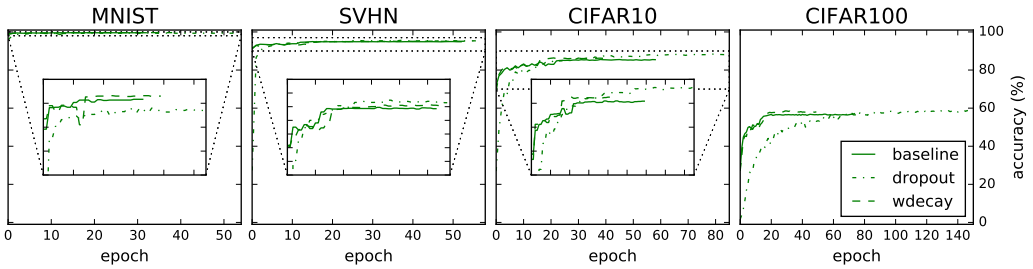


Figure 7.37: Validation accuracy comparison for CNN with ReLU activations and different regularization techniques.

In comparison with our baseline model, adding weight decay has a beneficial effect in all cases in terms of convergence speed and validation accuracy. Dropout, on the other hand, makes the model converge less quickly, a fact that makes sense intuitively, since the network only stochastically adjusts half its parameters in each update step. In some cases, namely SVHN and CIFAR10, the dropout model achieves higher validation accuracies, although it takes the model slightly longer to get there.

In conclusion, adding a small amount of weight decay is beneficial for regularization and to achieve slightly higher accuracies for model with or without batch normalization. Dropout shows some similarity to batch normalization in terms of gradient propagation and convergence behavior but slows down convergence rather drastically. Based on our experiments, adding dropout shows some of the same effects as batch normalization, specifically making the gradient fluctuations more well behaved over the course of training.

7.2.5 Batch size

Similar to the MLP experiments, we leave all hyperparameters constant but select the batch sizes in powers of two as in $\mathcal{B} = \{2^5, 2^6, ..., 2^{10}\}$. Figure 7.38 shows the

final test accuracies of the vanilla and batch normalized model after the last epoch of training when using different batch sizes.

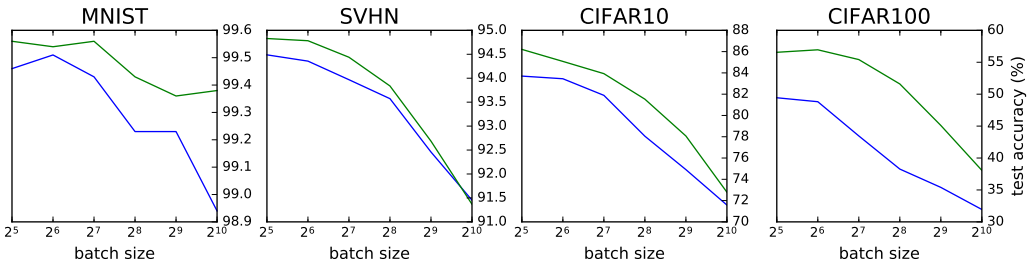


Figure 7.38: Test accuracies for CNN with ReLU activations using different batch sizes.

Our first observation is that the batch normalized model strictly outperforms its vanilla counterpart in terms of final test accuracy, independent of the batch size. The general trend is that smaller batch sizes outperform larger ones, whether batch normalization is used or not. Using this direct comparison, we conclude that the increased test accuracies for smaller batch sizes can be attributed to the noisier parameter updates in general and are not a direct effect of batch normalization.

Even more importantly, we cannot conclude that smaller batch sizes result in higher accuracies in general since the hyperparameters such as the initial learning rate and its decay have not been adjusted to the altered batch sizes. Similar to the MLP batch size experiments, we have a significantly larger number of individual parameter updates per batch with decreasing batch size.

Interestingly, as the batch size increases, the vanilla models suffer from slow convergence, especially in the beginning of training. Furthermore, we can observe severe loss fluctuations in the vanilla models, a behavior that the batch normalized model does not exhibit. Figure 7.39 visualizes these phenomena.

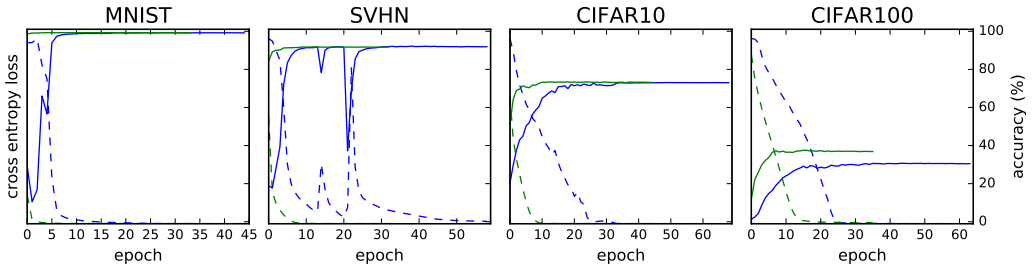


Figure 7.39: Training loss and validation accuracy for CNN with ReLU activations with batch size $\mathcal{B} = 1024$.

Looking at the gradient propagated throughout training of both models, we observe severe fluctuations of gradient magnitudes that coincide with the fluctuations

in the loss. Figure 7.40 shows the mean gradients for both models trained on all four datasets.

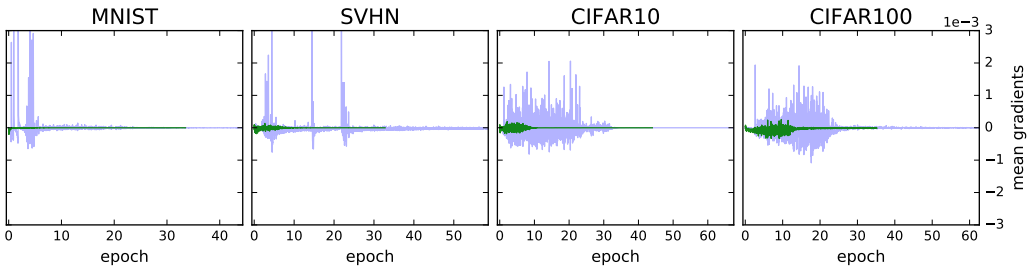


Figure 7.40: Mean gradients for CNN with ReLU activations with batch size $\mathcal{B} = 1024$.

Again, batch normalization reduces gradient magnitude fluctuations and makes the training better behaved. For all batch sizes, batch normalization achieves slightly higher test accuracies and speeds up convergence.

7.2.6 Epochs vs. wall time

Similar to the MLP experiments, we time each epoch for the vanilla and batch normalized CNN model. For our baseline model with a batch size of $\mathcal{B} = 64$, the average time per epoch for the vanilla model ranges from 87 to 128 seconds, and from 104 to 153 seconds for the batch normalized model. Thus, the average overhead for batch normalization is 19%, or 19 seconds per epoch. For a breakdown of wall time per epoch for different batch sizes, the reader is referred to the corresponding table B.4 in the appendix. Interestingly, for the CNN model, the overhead introduced by batch normalization decreases with larger batch sizes. This stems from the fact the the CNN model consists mainly of convolutional layers, which implement the batch normalization procedure differently than fully connected layers that are exclusively used in the MLP model.

Although the overhead created by adding batch normalization is lower for the CNN model and decreases for larger batch size, it still diminishes the convergence speed improvements slightly.

Chapter 8

Conclusion

This section contains some of the conclusions we can draw from our experiments and some more general practical recommendations when training neural networks. The following findings only hold for our experimental setup and we can not guarantee for them to be valid for other model architectures, image datasets, or hyperparameters. That being said, we strongly believe that our observations can be extrapolated to other models and datasets.

8.1 Batch Normalization

The following section summarizes the outcome of our experiments and presents the most important conclusions from applying batch normalization to multi-layer perceptrons and convolutional neural networks. For clarity, we make our conclusions dependent on specific hyperparameters and follow the structure of our experiments.

8.1.1 Activation functions

Considering activation functions, the most desirable property of batch normalization is that one can successfully train models with saturating nonlinearities like sigmoid, hyperbolic tangent, or softplus, which was not possible beforehand because of vanishing gradient problems. However, even when batch normalization is employed, the use of saturating activation functions is not encouraged since these models consistently achieve lower accuracies than the same model with rectified nonlinearities such as ReLU or ELU would.

Although ReLU activation functions have the undesirable property of dead neurons, they rightfully remain one of the most widely used nonlinearities for convolutional networks. From our observations, ReLU networks consistently achieve favorable results, while they do suffer from exploding gradients if batch normalization is not used. The exploding gradient problem for ReLU nonlinearities is usually handled via gradient clipping, a technique that manages the gradient magnitude

with the introduction of a threshold hyperparameter. The use of batch normalization alleviates the need to introduce such gradient clipping.

The theoretical properties of ELU nonlinearities seem superior at first since they are allegedly able to encode the absence of information as well as their presence. However, we cannot confirm that ELU networks significantly outperform ReLU networks with batch normalization. In fact, based our experiments, batch normalized ReLU networks match or outperform both vanilla and batch normalized ELU networks on all datasets and models. Moreover, we cannot confirm that batch normalization does not improve ELU networks since batch normalization either matches or slightly improves ELU networks on all datasets for the MLP model, and on half the datasets for the CNN model. To summarize, we cannot confirm any of the claims made by [5] about batch normalization and ELU activation functions, although we acknowledge that different results may have been obtained with different network architectures and hyperparameters.

From an intuitive standpoint, we conjecture that ReLU activations can also model the absence of information flowing through the network implicitly by not propagating negative values at all. By examining the per-layer activations of the CNN model, we can observe increasingly negative activations as the network depth progresses. When ReLU nonlinearities are used, these negative values cause the network to output zeros in most cases, which results in a sparse representation of the input data. If negative values are propagated using activation functions such as ELU, we can observe slightly more diffuse activations. However, the precise interpretation of these properties is beyond the scope of our work.

We have not included activation functions such as the leaky or parametric ReLU in our experiments mostly since the comparison of nonlinearities was not the main focus. However, we expect the parametric ReLU in particular to have favorable properties, especially since its negative slope can be adjusted by the model with backpropagation.

8.1.2 Initial learning rate

We can confirm that batch normalization enables significantly higher learning rates without the risk of divergence or other ill side effects. However, the main reason why batch normalization may accelerate deep network training is not the procedure itself, but rather that higher learning rates may be used since they do not cause the network to become unstable. Furthermore, faster learning rate decay might be another source of increased convergence speed.

Unfortunately, the authors of [28] do not compare their vanilla architecture with the batch normalized model when higher learning rates are used. Therefore, they do not show that simply adding batch normalization leads to significantly higher convergence speed if the other hyperparameters remain unchanged. It follows that their 14× convergence speed improvement is not primarily caused by batch normalization, but rather their adaptation of several hyperparameter such as the initial learning rate and its decay.

Interestingly, when the initial learning rate is decreased rather than increased, the effect of the batch normalization technique diminishes almost entirely, and we cannot confirm any significant speed or generalization improvements. Thus, our recommendation is to favor higher learning rates rather than lower ones when batch normalization is used.

It is possible that high learning rates that cause a vanilla model to diverge can be used with a batch normalized model without ill side effects. We argue that this increased learning rate and its accelerated decay is the main reason for the convergence speed improvements gained by [28]. However, we could not verify similar speed improvements when comparing our baseline models to ones with increased learning rates.

In our experiments, we have not altered the learning rate decay hyperparameter, nor cross-validated optimal decay rates for different datasets since adding another layer of complexity would have not fit the time frame of this work. However, we acknowledge that altering the learning rate decay may have serious consequences for the model convergence, and ultimately its generalization performance.

Throughout our experiments, we have used stochastic gradient descent with Nesterov momentum, which is a first-order optimization technique with favorable convergence properties. We acknowledge that adaptive optimization techniques such as Adam or RMSProp could have lead to different results, but at the cost of decreased interpretability. Batch normalization is a optimization-agnostic method, meaning that its benefits persist in conjunction with both global and adaptive optimization methods. Our initial experiments have confirmed this.

8.1.3 Weight initialization

When batch normalization is used, one can initialize the weights to almost arbitrary values as long as the symmetry of the weights is broken. This is generally achieved by drawing the initial weights randomly from a normal or uniform distribution. Although being able to train networks from almost arbitrary initial parameters is desirable, we find that the variance preserving initialization schemes also show their theoretical superiority in practice, which is why we recommend the use of Kaiming initialization for rectifiers and Xavier initialization for saturating nonlinearities.

Since the gradients flowing through the network are inversely proportional to the scale of the weights, we find that networks initialized with smaller random values generally lead to larger gradients and thus faster convergence. However, we cannot conclude that smaller variance initialization strictly leads to favorable generalization performance.

8.1.4 Regularization

We can confirm that batch normalization acts as a regularizer that exhibits some of the same properties of dropout, without slowing down the convergence of the network. The main similarity of batch normalization and dropout is that the magnitude

of the gradients remains stable throughout the whole training process, at least until the network approaches near-zero losses. We find that the gradients do not vanish even after the dropout model loss approaches zero. Intuitively, this makes sense because the dropout network only utilized half its potential activation during each forward and backward pass, thus the stochasticity prevents the loss from reaching absolute zero. In case of using batch normalization, the regularizing effect likely stems from the fact that individual examples are normalized over entirely different batches containing different examples for each parameter update.

We experimentally verify that vanilla models with dropout generalize better to unseen data than batch normalized models with dropout. We conjecture that dropout counteracts some of the beneficial effects of batch normalization by not accounting for the stochastically dropped activations.

We find that adding a small amount of L_2 weight regularization to the loss helps the model generalize regardless of other hyperparameters. We cannot conclude that the regularization strength has to be reduced in order to gain higher accuracies from batch normalization. For vanilla networks, we argue that weight regularization can help with exploding gradients and might resolve the need for gradient clipping.

8.1.5 Batch size

Interestingly, smaller batch sizes work strictly better for batch normalized MLP models, whereas they worsen the predictive power of their vanilla counterparts. For smaller batch sizes, we mainly attribute this behavior to the vanilla model's tendency to overfit when trained too long, which puts the use of batch normalization in a more favorable light since those types of models do not overfit as severely. On the other hand, for larger batch sizes, batch normalization negatively affects the model's generalization performance and is a strong case against the technique. We could not find a conclusive reason why this is the case.

The relative improvement gained from batch normalization in case of the CNN model seems to be unrelated to the choice of batch size, whereas the absolute generalization performance improves with smaller batch sizes. Just by adding batch normalization to our CNN model, we can achieve higher validation and test accuracies on all datasets and batch sizes, which is not the case for the MLP model. Therefore, we argue that batch normalization is better suited to work with convolutional layers than it is with fully connected layers. Additionally, the time overhead for employing batch normalization in convolutional layers seems to scale more effectively with larger batch sizes, which is especially desirable as the working memory of the underlying hardware grows.

However, we have to note that our training methodology did not change when different batch sizes were used and the remaining hyperparameters have not been cross-validated with respect to the change in batch size. This leads to the conclusion that larger batch sizes may also increase the convergence speed and generalization performance of both types of models, provided that all other hyperparameters have been adjusted to reflect the change in batch size. We find it hard to favor a particular

choice of batch size, although our experiments suggest smaller batches as advantageous. Ultimately, the upper bound on the batch size is given by the amount of memory that the model uses to process the batch, whereas the lower bound is given by how long one wants to wait for the model to process it.

8.1.6 Epochs vs. wall time

Although batch normalization adds a certain amount of computational complexity that results in slightly higher training times, we generally find that the benefits outweigh the introduced overhead. In fact, for our convolutional neural network architecture, the overhead is negligible and scales nicely with larger batch sizes. The opposite is true for the multi-layer perceptron, where we find that batch normalization adds relatively more overhead with increasing the batch size.

To further reduce the overhead introduced by batch normalization, one can distribute the normalization layers more sparsely throughout the model architecture. We suspect that reducing the number of batch normalization layers may lead to similar benefits, while reducing their computational footprint.

8.2 Practical recommendations

Based on our experiments, we can establish that the initial learning rate is in fact the most important hyperparameter in neural network training. We agree with the recommendation provided in [2] that one should pick the largest possible learning rate that does not cause the model to diverge. Batch normalization enables us to sample possible values for the initial learning rate from a larger distribution.

After the initial learning rate is chosen, the next crucial hyperparameter is the learning rate decay. In our experiments, we found that adaptively decaying the learning rate based on the validation accuracy measured after each epoch performs strictly better than exponential or power decay. Naturally, one can find optimal parameters for power and exponential decay with cross validation, but decaying the learning rate based on the validation accuracy is an intuitive heuristic that works very well in practice.

Regarding weight initialization, we recommend the use of variance preserving initialization schemes such as the ones discussed in chapter 2, whether batch normalization is used or not. Specifically we recommend using Kaiming initialization for rectified activations such as ReLU and ELU. Although saturating nonlinearities are not recommended, one should favor Xavier initialization for sigmoids and hyperbolic tangents.

8.3 Closing remarks

Batch normalization can alleviate many of the common pitfalls in neural network training. Increased learning rates that would cause a vanilla model to diverge can

be beneficial in batch normalized models and even accelerate their convergence. In most experiments, models using batch normalization outperformed their vanilla counterparts. However, since the best accuracies in our experiments were achieved by a vanilla state-of-the-art model using dropout, we can not unanimously recommend the use of batch normalization in practice.

Our advice to the neural network practitioner is to experiment with deep convolutional architectures that employ dropout, variance preserving weight initialization, rectified nonlinearities, weight decay, and adaptive optimization techniques. In general, monitoring the loss and validation accuracy, as well as the layer activations and gradient propagation can help to isolate and address problems early in training. If the model shows signs of slow convergence or even divergence, we recommend to add batch normalization layers before each nonlinearity in order to counteract the poor behavior.

In conclusion, batch normalization is an exciting addition to the neural network toolbox that shows very promising characteristics. While the idea of normalizing activations throughout the model is arguably simple, its effects are profound and show how small architectural improvements can make all the difference in the field of deep learning. Another example of this is the move from saturating to rectified nonlinearities, which enabled deep neural networks to learn effectively without the problem of vanishing gradients. This conceivably small change was partly responsible for the renewed interest in neural networks with many processing layers after years of negligence.

With the rising interest in artificial intelligence and deep learning research today, we expect to see numerous advances in these fields in the near future. Although our quest to solve intelligence may be in its infancy, the pace of technological progress is nothing short of astounding. In order to end this thesis on a positive note, the reader is referred to appendix C for a discussion on the ethical impact of our work.

Appendix A

Extended experimental setup

In the following sections, we provide an overview of both the software and hardware used to conduct the experiments presented in this work.

A.1 Software

The software framework used for conducting experiments is Torch [6], a scientific computing platform with wide support for machine learning algorithms that is used and maintained by various technology companies such as Google DeepMind and Facebook AI Research. At its core, Torch supplies a flexible n-dimensional array with many useful routines such as indexing, slicing and transposing. Moreover, Torch features an extremely fast scripting language and is based on LuaJIT, a powerful just-in-time compiler. LuaJIT is implemented in the C programming language and thus provides a clean interface to the GPU using NVIDIA's CUDA libraries.

CUDA is a parallel computing platform and application programming interface that enables general purpose GPU processing. In this research, we make heavy use of both the CUDA Toolkit 7.5 and cuDNN v4, a deep neural network library that extends the toolkit with useful operations such as highly optimized convolutions. An exciting piece of trivia is that NVIDIA advertises cuDNN v4 with the following slogan: "Train neural networks up to 14x faster with batch normalization" [1].

One of Torch's main advantages is the availability of packages for deep learning applications that use CUDA and cuDNN as a backend. For instance, the `nn` package provides a neural network library that comes with many useful layers such as `SpatialConvolution`, `BatchNormalization`, and `ReLU`. The `optim` package for numerical optimization comes with algorithms such as `SGD`, `Adagrad`, and `Adam`.

Torch's API bears significant resemblance to the pseudocode used in the backpropagation algorithm (algorithm 1) since each module provides a `forward` and `backward` function that implements the mathematical notation introduced in sections 2.9.1, 2.9.2, 2.9.3, and 3.

[1] www.developer.nvidia.com/cudnn

A.2 Hardware

The machine used for the experiments features a 3.4GHz Intel Core i7-2600K with 16GB of DDR3 RAM, and a NVIDIA Tesla K40c GPU with 2880 CUDA cores clocked at 745MHz and 12GB of RAM. The system is running Ubuntu 14.04.1 LTS with GNU/Linux kernel 3.19.0-51.

A.2.1 Acknowledgements

We gratefully acknowledge the support of NVIDIA Corporation with the donation of a Tesla K40 GPU used for this research.

Appendix B

Extended experimental results

Here I will include the tables that contain the best validation accuracies and final test accuracies for all conducted experiments.

B.1 Test accuracy

B.1.1 Multi-layer perceptron

Table B.1 provides an overview of the final accuracy on the test set for the MLP model after the last epoch of training.

		MNIST		**SVHN**		**CIFAR10**		**CIFAR100**	
Parameter	**Value**	V	B	V	B	V	B	V	B
Activation function	sigmoid	97.4	**97.9**	80.0	**82.6**	45.4	**49.8**	15.2	**21.9**
	ReLU	98.0	**98.2**	81.4	**84.1**	49.6	**51.6**	22.7	**24.2**
	ELU	98.0	**98.2**	80.9	**83.9**	50.4	**51.7**	21.8	**23.9**
Initial learning rate	high	11.4	**98.6**	19.6	**83.6**	10.0	**52.6**	1.0	**25.5**
	low	97.6	**97.7**	82.8	**83.2**	49.0	**49.3**	**23.1**	20.7
Weight initialization	high	9.8	**96.3**	19.6	**76.4**	10.0	**46.1**	1.0	**15.8**
	low	11.4	**98.4**	19.6	**84.5**	10.0	**52.6**	1.0	**24.7**
Regularization	dropout	96.8	**97.6**	75.4	**77.4**	48.0	**50.3**	15.6	**16.9**
	w. decay	98.2	**98.4**	83.4	**84.9**	51.0	**51.8**	23.2	**25.0**

Table B.1: MLP test accuracies for different experiments in percent.

B.1.2 Convolutional neural network

Table B.2 provides an overview of the final accuracy on the test set for the CNN model after the last epoch of training.

Parameter	Value	MNIST		SVHN		CIFAR10		CIFAR100	
		V	B	V	B	V	B	V	B
Activation function	sigmoid	11.4	**99.4**	19.6	**91.8**	10.0	**77.3**	1.0	**40.3**
	ReLU	99.5	**99.5**	94.4	**94.8**	83.5	**85.1**	48.8	**57.0**
	ELU	99.5	**99.5**	**94.2**	93.6	**83.4**	82.9	53.9	**55.5**
Initial learning rate	high	11.4	**99.6**	19.6	**95.2**	10.0	**86.5**	1.0	**56.2**
	low	99.2	**99.4**	91.8	**93.1**	76.4	**78.1**	**35.3**	46.2
Weight initialization	high	9.8	**98.4**	19.6	**86.5**	10.0	**62.0**	1.0	**29.8**
	low	11.4	**99.6**	19.6	**94.8**	10.0	**85.0**	1.0	**51.9**
Regularization	dropout	**99.6**	99.5	**96.1**	96.1	**87.8**	87.3	58.0	**58.5**
	w. decay	99.4	**99.6**	94.2	**95.2**	83.5	**86.0**	49.1	**59.1**

Table B.2: CNN test accuracies for different experiments in percent.

B.2 Wall time

B.2.1 Multi-layer perceptron

Table B.3 provides an overview of the average time spend per epoch for the MLP model. Generally, the overhead for batch normalization increases with increasing batch size.

Batch size	Mean overhead	MNIST		SVHN		CIFAR10		CIFAR100	
		V	B	V	B	V	B	V	B
32	20%	2.6	3.1	3.7	4.4	2.5	3.0	2.5	3.1
64	27%	1.8	2.4	2.7	3.4	1.8	2.3	1.9	2.4
128	35%	1.4	2.0	2.2	2.9	1.5	2.0	1.5	2.0
256	38%	1.2	1.8	2.0	2.7	1.4	1.8	1.4	1.9
512	39%	1.2	1.7	1.9	2.6	1.3	1.7	1.3	1.8
1024	41%	1.1	1.7	1.8	2.5	1.2	1.7	1.2	1.7

Table B.3: MLP average time per epoch for different batch sizes in seconds.

B.2.2 Convolutional neural network

Table B.4 provides an overview of the average time spend per epoch for the CNN model. Generally, the overhead for batch normalization decreases with increasing batch size.

Batch size	**Mean overhead**	**MNIST**		**SVHN**		**CIFAR10**		**CIFAR100**	
		V	B	V	B	V	B	V	B
32	19.7%	141	162	166	201	112	137	113	137
64	19.0%	103	122	128	153	87	104	87	104
128	16.4%	86	101	107	124	73	85	73	85
256	15.5%	77	89	95	109	65	75	65	75
512	15.3%	72	83	89	102	60	69	60	70
1024	15.2%	69	80	86	99	58	66	58	66

Table B.4: CNN average time per epoch for different batch sizes in seconds.

Appendix C

Ethical considerations

In recent years, artificial intelligence (AI) and machine learning (ML) technologies have seen an enormous rise in popularity and continue to gain momentum in academia, in the industry, and in society as a whole. For example, many tech companies such as Google, Facebook, or Amazon are actively promoting the use of AI technologies in their products to enhance people's lives. The question of whether or not these technologies have a positive impact is at the heart of a controversial debate that is taking place today. Many influential researchers and business executives such as Stephen Hawking, Bill Gates, and Elon Musk have publicly expressed their concern with regards to the future of AI in an open letter[1]. The document affirms that society can obtain great potential benefits from AI, but calls for concrete research on how to prevent certain potential risks. The letter has been signed by nearly ten thousand people as of this writing.

The potential benefits and risks of artificial intelligence are numerous and the following should be seen as a high level overview that facilitates an understanding of how our work in particular fits into the debate.

On the one hand, artificial intelligence can be seen as an extension of human capabilities that can help us solve many real-world problems or simply make our lives easier and more convenient. A popular example of how AI systems can achieve this is the health care industry. As of this writing, machine learning algorithms are already supporting doctors and health care practitioners in the prognosis and prediction of fatal illnesses such as cancer or Alzheimer's disease by sifting through vast amounts of medical data. For example, a substantial part of this data consists of noisy medical imagery that can be fed to a computer vision system that is able to localize regions of interest, possibly detecting and classifying tumors. Automating these tasks with machine learning can help to identify symptoms early, treat patients effectively, and perhaps save lives in the process.

On the other hand, very similar technology as in the previous example can be abused in many harmful ways, the most popular examples being automated surveillance and warfare. The technology that locates tumors in medical images

[1] `http://futureoflife.org/ai-open-letter/`

can potentially be used for detecting humans and classifying faces in surveillance camera footage. To make matters worse, one can imagine an armed drone that is sent into the battlefield to not only localize human targets, but to kill them.

As we can see from the two previous examples, the same underlying technology can be used for image classification in medical data or human target localization in automated warfare. Convolutional neural networks, as introduced in our work, provide a framework that is able to solve both of these tasks very effectively. As an attempt to bridge the gap between these examples and the work presented in this thesis, batch normalization has the potential to make both medical image classification and human target localization more accurate. It is ultimately not the model itself that bears the risk, but the intention with which it is used.

From an economical point of view, artificially intelligent systems bear the risk of making many jobs in our society redundant. As of today, many repetitive tasks can be automated and we see robots replacing even cognitively challenging jobs as the technology progresses. For instance, computer vision systems are already substituting human employees that analyze surveillance camera footage or further down the line autonomous cars may may replace taxi drivers entirely. On the other side of the coin, robots are essentially protecting humans from injuries in high risk environments such as factories and autonomous cars may have huge impact on safety on the road by preventing traffic accidents.

From a societal perspective, computer vision technology powers many aspects of our daily lives such as image based web searches, automatically tagging people in pictures on social media platforms, and categorizing photos in our digital photo albums. However, apart from the added convenience of these features, the created metadata can be used to track what we are searching online, create graphs of our social circle, or invade our privacy by analyzing the content of our photos.

As we can see from the previous examples, there is an inherent moral duality to artificial intelligence technologies. Systems that can be used to protect and enhance people's lives can alose be used to invade our privacy or harm us physically. We have to keep in mind that with great power comes great responsibility. In conclusion, we believe in the great potential that artificial intelligence and machine learning offers, and hope that these models are used exclusively for the purpose of enhancing our lives and help humankind in solving real-world problems.

Bibliography

[1] D. Arpit, Y. Zhou, B. U. Kota, and V. Govindaraju. Normalization Propagation: A Parametric Technique for Removing Internal Covariate Shift in Deep Networks. *ArXiv e-prints*, 2016.

[2] Y. Bengio. Practical Recommendations for Gradient-based Training of Deep Architectures. *CoRR*, abs/1206.5533, 2012.

[3] J. Bergstra and Y. Bengio. Random Search for Hyper-parameter Optimization. *J. Mach. Learn. Res.*, 13(1):281–305, 2012.

[4] D. C. Ciresan, U. Meier, and J. Schmidhuber. Multi-column Deep Neural Networks for Image Classification. *CoRR*, 2012.

[5] D. Clevert, T. Unterthiner, and S. Hochreiter. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *CoRR*, 2015.

[6] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*, 2011.

[7] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. P. Kuksa. Natural Language Processing (almost) from Scratch. *CoRR*, 2011.

[8] T. Cooijmans, N. Ballas, C. Laurent, Ç. Gülçehre, and A. Courville. Recurrent Batch Normalization. *ArXiv*, 2016.

[9] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[10] E. L. Denton, S. Chintala, a. szlam, and R. Fergus. Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 1486–1494. Curran Associates, Inc., 2015.

[11] J. Duchi, E. Hazan, and Y. Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *J. Mach. Learn. Res.*, 12:2121–2159, 2011.

[12] L. Fei-Fei, A. Karpathy, and J. Johnson. CS231n: Convolutional Neural Networks for Visual Recognition. 2016.

[13] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. *CoRR*, 2013.

[14] X. Glorot and Y. Bengio. Understanding the Difficulty of Training Deep Feedforward Neural Networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[15] X. Glorot, A. Bordes, and Y. Bengio. Deep Sparse Rectifier Neural Networks. In G. J. Gordon and D. B. Dunson, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)*, volume 15, pages 315–323. Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011.

[16] I. Goodfellow, Y. Bengio, and A. Courville. Deep Learning. Book in preparation for MIT Press, 2016.

[17] B. Graham. Fractional Max-Pooling. *CoRR*, 2014.

[18] Ç. Gülçehre and Y. Bengio. Knowledge Matters: Importance of Prior Information for Optimization. *CoRR*, abs/1301.4083, 2013.

[19] S. Gupta, R. B. Girshick, P. Arbelaez, and J. Malik. Learning Rich Features from RGB-D Images for Object Detection and Segmentation. *CoRR*, 2014.

[20] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *CoRR*, 2015.

[21] K. He, X. Zhang, S. Ren, and J. Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *CoRR*, 2015.

[22] G. Hinton, L. Deng, D. Yu, A. rahman Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. S. G. Dahl, and B. Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[23] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Improving Neural Networks by Preventing Co-adaptation of Feature Detectors. *CoRR*, 2012.

[24] S. Hochreiter. Untersuchungen zu Dynamischen Neuronalen Netzen. *Master's thesis, Institut für Informatik, Technische Universität, München*, 1991.

[25] D. H. Hubel and T. N. Wiesel. Receptive Fields of Single Neurones in the Cat's Striate Cortex. *The Journal of Physiology*, 148(3):574–591, 1959.

[26] D. H. Hubel and T. N. Wiesel. Receptive Fields, Binocular Interaction and Functional Architecture in the Cat's Visual Cortex. *The Journal of physiology*, 160:106–154, 1962.

[27] D. H. Hubel and T. N. Wiesel. Receptive Fields and Functional Architecture of Monkey Striate Cortex. *The Journal of Physiology*, 195(1):215–243, 1968.

[28] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*, 2015.

[29] M. Jaderberg, K. Simonyan, A. Zisserman, and k. kavukcuoglu. Spatial Transformer Networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2008–2016. Curran Associates, Inc., 2015.

[30] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the Best Multi-Stage Architecture for Object Recognition? In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 2146–2153, 2009.

[31] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *CoRR*, 2014.

[32] A. Krizhevsky and G. Hinton. Learning Multiple Layers of Features from Tiny Images, 2009.

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[34] C. Laurent, G. Pereyra, P. Brakel, Y. Zhang, and Y. Bengio. Batch Normalized Recurrent Neural Networks. *ArXiv*, 2015.

[35] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back. Face Recognition: A Convolutional Neural-Network Approach. *IEEE Transactions on Neural Networks*, 8(1):98–113, 1997.

[36] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Comput.*, 1(4):541–551, 1989.

[37] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based Learning Applied to Document Recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[38] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Effiicient BackProp. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, pages 9–50, London, UK, UK, 1998. Springer-Verlag.

[39] Y. Lecun and C. Cortes. The MNIST Database of Handwritten Digits. 1998.

[40] C.-Y. Lee, P. W. Gallagher, and Z. Tu. Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree. *arXiv preprint arXiv:1509.08985*, 2015.

[41] Z. Liao and G. Carneiro. On the Importance of Normalisation Layers in Deep Learning with Piecewise Linear Activation Units. *CoRR*, abs/1508.00330, 2015.

[42] J. Long, E. Shelhamer, and T. Darrell. Fully Convolutional Networks for Semantic Segmentation. *CoRR*, 2014.

[43] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading Digits in Natural Images with Unsupervised Feature Learning. *NIPS workshop on deep learning and unsupervised feature learning*, 2011:4, 2011.

[44] H. Noh, S. Hong, and B. Han. Learning Deconvolution Network for Semantic Segmentation. In *The IEEE International Conference on Computer Vision (ICCV)*, 2015.

[45] R. Pascanu, T. Mikolov, and Y. Bengio. On the Difficulty of Training Recurrent Neural Networks. *CoRR*, abs/1211.5063, 2012.

[46] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *CoRR*, 2015.

[47] S. Ren, K. He, R. B. Girshick, and J. Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *CoRR*, 2015.

[48] M. Riedmiller and H. Braun. A Direct Adaptive Method for Faster Backpropagation Learning: the RPROP Algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591 vol.1, 1993.

[49] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning Representations by Back-Propagating Errors. *Cognitive modeling*, 5(3):1, 1988.

[50] T. Salimans and D. P. Kingma. Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks. *CoRR*, abs/1602.07868, 2016.

[51] A. M. Saxe, J. L. McClelland, and S. Ganguli. Exact Solutions to the Nonlinear Dynamics of Learning in Deep Linear Neural Networks. *CoRR*, 2013.

[52] H. Shimodaira. Improving Predictive Inference Under Covariate Shift by Weighting the Log-likelihood Function. *Journal of Statistical Planning and Inference*, 90(2):227 – 244, 2000.

[53] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*, 2014.

[54] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. A. Riedmiller. Striving for Simplicity: The All Convolutional Net. *CoRR*, 2014.

[55] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[56] R. Stewart and M. Andriluka. End-to-end People Detection in Crowded Scenes. *CoRR*, 2015.

[57] I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton. On the Importance of Initialization and Momentum in Deep Learning. In *ICML (3)*, volume 28 of *JMLR Proceedings*, pages 1139–1147. JMLR.org, 2013.

[58] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going Deeper with Convolutions. *CoRR*, 2014.

[59] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the Inception Architecture for Computer Vision. *CoRR*, abs/1512.00567, 2015.

[60] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '14, pages 1701–1708, Washington, DC, USA, 2014. IEEE Computer Society.

[61] T. Tijmen and G. Hinton. Lecture 6.5: RMSProp. Coursera: Neural Networks for Machine Learning, 2013.

[62] A. Vedaldi and K. Lenc. MatConvNet: Convolutional Neural Networks for MATLAB. In *Proceedings of the 23rd ACM International Conference on Multimedia*, MM '15, pages 689–692, New York, NY, USA, 2015. ACM.

[63] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and Tell: A Neural Image Caption Generator. *CoRR*, 2014.

[64] S. Wager, S. Wang, and P. S. Liang. Dropout Training as Adaptive Regularization. In C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 351–359. Curran Associates, Inc., 2013.

[65] L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus. Regularization of Neural Networks using DropConnect. In S. Dasgupta and D. Mcallester, editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 1058–1066. JMLR Workshop and Conference Proceedings, 2013.

[66] S. Wang and C. Manning. Fast Dropout Training. In S. Dasgupta and D. Mcallester, editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 118–126. JMLR Workshop and Conference Proceedings, 2013.

[67] B. Xu, N. Wang, T. Chen, and M. Li. Empirical Evaluation of Rectified Activations in Convolutional Network. *CoRR*, 2015.

[68] M. D. Zeiler. Adadelta: An Adaptive Learning Rate Method. *CoRR*, 2012.