CHAPTER 5

CNN Learning

In chapter 4, we discussed different architecture blocks of the CNN and their operational details. Most of these CNN layers involve parameters which are required to be tuned appropriately for a given computer vision task (e.g., image classification and object detection). In this chapter, we will discuss various mechanisms and techniques that are used to set the weights in deep neural networks. We will first cover concepts such as weight initialization and network regularization in Sections 5.1 and 5.2 respectively, which helps in a successful optimization of the CNNs. Afterwards, we will introduce gradient based parameter learning for CNNs in Section 5.3, which is quite similar to the MLP parameter learning process discussed in Chapter 3. The details of neural network optimization algorithms (also called '*solvers*') will come in Section 5.4. Finally in Section 5.5, we will explain various types of approaches which are used for the calculation of the gradient during the error back-propagation process.

5.1 WEIGHT INITIALIZATION

A correct weight initialization is the key to stably train very deep networks. An ill-suited initialization can lead to the vanishing or exploding gradient problem during error back-propagation. In this section, we introduce several approaches to perform weight initialization and provide comparisons between them to illustrate their benefits and problems. Note that the discussion below pertains to the initialization of neuron weights within a network and the biases are usually set to zero at the start of the network training. If all the weights are also set to zero at the start of training, the weight updates will be identical (due to symmetric outputs) and the network will not learn anything useful. To break this symmetry between neural units, the weights are initialized randomly at the start of the training. In the following, we describe several popular approaches to network initialization.

5.1.1 GAUSSIAN RANDOM INITIALIZATION

A common approach to weight initialization in CNNs is the Gaussian random initialization technique. This approach initializes the convolutional and the fully connected layers using random matrices whose elements are sampled from a Gaussian distribution with zero mean and a small standard deviation (e.g., 0.1 and 0.01).

5.1.2 UNIFORM RANDOM INITIALIZATION

The uniform random initialization approach initializes the convolutional and the fully connected layers using random matrices whose elements are sampled from a uniform distribution (instead of a normal distribution as in the earlier case) with a zero mean and a small standard deviation (e.g., 0.1 and 0.01). The uniform and normal random initializations generally perform identically. However, the training of very deep networks becomes a problem with a random initialization of weights from a uniform or normal distribution [Simonyan and Zisserman, 2014]. The reason is that the forward and back-ward propagated activations can either diminish or explode when the network is very deep (see Section 3.2.2).

5.1.3 ORTHOGONAL RANDOM INITIALIZATION

Orthogonal random initialization has also been shown to perform well in deep neural networks [Saxe et al., 2013]. Note that the Gaussian random initialization is only approximately orthogonal. For the orthogonal random initialization, a random weight matrix is decomposed by applying e.g., a Singular Value Decomposition (SVD). The orthogonal matrix (U) is then used for the weight initialization of the CNN layers.

5.1.4 UNSUPERVISED PRE-TRAINING

One approach to avoid the gradient diminishing or exploding problem is to use layer-wise pre-training in an unsupervised fashion. However, this type of pre-training has found more success in the training of deep generative networks e.g., Deep Belief Networks [Hinton et al., 2006] and Auto-encoders [Bengio et al., 2007]. The unsupervised pre-training can be followed by a supervised fine-tuning stage to make use of any available annotations. However, due to the new hyper-parameters, the considerable amount of effort involved in such an approach and the availability of better initialization techniques, layer-wise pre-training is seldom used now to enable the training of CNN based very deep networks. We describe some of the more successful approaches to initialize deep CNNs next.

5.1.5 XAVIER INITIALIZATION

A random initialization of a neuron makes the variance of its output directly proportional to the number of its incoming connections (a neuron's fan-in measure). To alleviate this problem, Glorot and Bengio [2010] proposed to randomly initialize the weights with a variance measure that is dependent on the number of incoming and outgoing connections (n_{f-in} and n_{f-out} respectively) from a neuron,

$$Var(w) = \frac{2}{n_{f-in} + n_{f-out}}, \tag{5.1}$$

where w are network weights. Note that the fan-out measure is used in the variance above to balance the back-propagated signal as well. Xavier initialization works quite well in practice

and leads to better convergence rates. But a number of simplistic assumptions are involved in the above initialization, among which the most prominent is that a linear relationship between the input and output of a neuron is assumed. In practice all the neurons contain a non-linearity term which makes Xavier initialization statistically less accurate.

5.1.6 RELU AWARE SCALED INITIALIZATION

He et al. [2015a] suggested an improved version of the scaled (or Xavier) initialization noting that the neurons with a ReLU non-linearity do not follow the assumptions made for the Xavier initialization. Precisely, since the ReLU activation reduces nearly half of the inputs to zero, therefore the variance of the distribution from which the initial weights are randomly sampled should be,

$$Var(w) = \frac{2}{n_{f-in}}. \tag{5.2}$$

The ReLU aware scaled initialization works better compared to Xavier initialization for recent architectures which are based on the ReLU nonlinearity.

5.1.7 LAYER-SEQUENTIAL UNIT VARIANCE

The Layer-sequential unit variance (LSUV) initialization is a simple extension of the orthonormal weight initialization in deep network layers [Mishkin and Matas, 2015]. It combines the benefits of batch-normalization and the orthonormal weight initialization to achieve an efficient training for very deep networks. It proceeds in two steps, described below:

- **Orthogonal initialization** - In the first step, all the weight layers (convolutional and fully connected) are initialized with orthogonal matrices.
- **Variance normalization** - In the second step, the method starts from the initial towards the final layers in a sequential manner and the variance of each layer output is normalized to one (unit variance). This is similar to the batch normalization layer, which normalizes the output activations for each batch to be zero centered with a unit variance. However, different from batch normalization which is applied during the training of the network, LSUV is applied while initializing the network and therefore saves the overhead of normalization for each batch during the training iterations.

5.1.8 SUPERVISED PRE-TRAINING

In practical scenarios, the training of very deep networks is desired, but very often we do not have a large amount of data available for a specific problem setting. A good practice in such cases is to first train the network on a related problem, where a large amount of training data is available. Afterwards, the learned model can be 'adapted' to a new task by

initializing with weights pre-trained on a larger dataset. This process is called *fine-tuning* and is a simple, yet an effective way to transfer learning from one task to another (domain transfer or domain adaptation). As an example, in order to perform scene classification on a relatively small dataset, MIT-67, the network can be initialized with the weights learned for object classification on the much larger ImageNet dataset [Khan et al., 2016b].

5.2 REGULARIZATION OF CNN

Since deep neural networks have a large number of parameters, they tend to over-fit on the training data during the learning process. By over-fitting, we mean that the model performs really well on the training data but it fails to generalize well to unseen data. It, therefore, results in an inferior performance on new data (usually the test set). Regularization approaches aim to avoid this problem using several intuitive ideas which we discuss below. We can categorize common regularization approaches into the following classes, based on their central idea:

- Approaches which regularize the network using data level techniques (e.g., data augmentation),
- Approaches which introduce stochastic behavior in the neural activations (e.g., dropout and drop connect),
- Approaches which normalize batch statistics in the feature activations (e.g., batch normalization),
- Approaches which use decision level fusion to avoid over-fitting (e.g., ensemble model averaging),
- Approaches which introduce constraints on the network weights (e.g., ℓ^1 norm, ℓ^2 norm, max-norm and elastic net constraints),
- Approaches which use guidance from a validation set to halt the learning process (e.g., early stopping).

Next, we discuss the above mentioned approaches in detail.

5.2.1 DATA AUGMENTATION

Data augmentation is the easiest, and often a very effective way of enhancing the generalization power of CNN models. Especially for cases where the number of training examples is relatively low, data augmentation can enlarge the dataset (by factors of 16x, 32x, 64x or even more) to allow a more robust training of large-scale models.

Data augmentation is performed by making several copies from a single image using straightforward operations such as rotations, cropping, flipping, scaling, translations

and shearing (see Figure 5.1). These operations can be performed separately or combined together to form copies e.g., which are both flipped and cropped.

Color jittering is another common way of performing data augmentation. A simple form of this operation is to perform random contrast jittering in an image. One could also find the principal color directions in the R, G and B channels (using PCA) and then apply a random offset along these directions to change the color values of the whole image. This effectively introduces color and illumination invariance in the learned model [Krizhevsky et al., 2012].

Another approach for data augmentation is to utilize synthetic data, alongside the real data, to improve the generalization ability of the network [Rahmani and Mian, 2016, Rahmani et al., 2017, Shrivastava et al., 2016]. Since, synthetic data is usually available in large quantities from rendering engines, it effectively extends the training data, which helps avoid over-fitting.

Figure 5.1: The figure shows an example of data augmentation using crops (column 1 and 2), rotations (column 3) and flips (column 4). Since the input image is quite complex (has several objects), data augmentation allows the network to figure out some possible variations of the same image, which still denote the same scene category i.e., a theater.

5.2.2 DROPOUT

One of the most popular approaches for neural network regularization is the Dropout technique [Srivastava et al., 2014]. During network training, each neuron is activated with a fixed probability (usually 0.5 or set using a validation set). This random sampling of a

sub-network within the full-scale network introduces an ensemble effect during the testing phase, where the full network is used to perform prediction. Activation dropout works really well for regularization purposes and gives a significant boost in performance on unseen data in the test phase.

Let us consider a CNN that is composed of L weight layers, indexed by $l \in \{1 \dots L\}$. Since Dropout has predominantly been applied to Fully Connected (FC) layers in the literature, we consider the simpler case of FC layers here. Given output activations $\mathbf{a}_{l-1}$ from the previous layer, a FC layer performs an affine transformation followed by a element-wise non-linearity, as follows:

$$\mathbf{a}_l = f(\mathbf{W} * \mathbf{a}_{l-1} + \mathbf{b}_l). \tag{5.3}$$

Here, $\mathbf{a}_{l-1} \in \mathbb{R}^n$ and $\mathbf{b} \in \mathbb{R}^m$ denote the activations and biases respectively. The input and output dimensions of the FC layer are denoted by n and m respectively. $\mathbf{W} \in \mathbb{R}^{m \times n}$ is the weight matrix and $f(\cdot)$ is the Rectified Linear Unit (ReLU) activation function.

The random dropout layer generates a mask $\mathbf{m} \in \mathbb{B}^m$, where each element m_i is independently sampled from a Bernoulli distribution with a probability 'p' of being 'on' i.e., a neuron fires:

$$m_i \sim Bernoulli(p), \qquad m_i \in \mathbf{m}. \tag{5.4}$$

This mask is used to modify the output activations $\mathbf{a}_l$:

$$\mathbf{a}_l = \mathbf{m} \circ f(\mathbf{W} * \mathbf{a}_{l-1} + \mathbf{b}_l), \tag{5.5}$$

where, '$\circ$' denotes the Hadamard product. The Hadamard product denotes a simple element wise matrix multiplication between the mask and the CNN activations.

5.2.3 DROP-CONNECT

Another similar approach to Dropout is the Drop-Connect [Wan et al., 2013], which randomly deactivates the network weights (or connections between neurons) instead of randomly reducing the neuron activations to zero.

Similar to Dropout, Drop-Connect performs a masking out operation on the weight matrix instead of the output activations, therefore:

$$\mathbf{a}_l = f((\mathbf{M} \circ \mathbf{W}) * \mathbf{a}_{l-1} + \mathbf{b}_l), \tag{5.6}$$

$$M_{i,j} \sim Bernoulli(p), \qquad M_{i,j} \in \mathbf{M}. \tag{5.7}$$

where,'$\circ$' denotes the Hadamard product as in the case of Dropout.

5.2.4 BATCH NORMALIZATION

Batch normalization [Ioffe and Szegedy, 2015] normalizes the mean and variance of the output activations from a CNN layer to follow a unit Gaussian distribution. It proves to

be very useful for the efficient training of a deep network because it reduces the 'internal covariance shift' of the layer activations. Internal covariance shift refers to the change in the distribution of activations of each layer as the parameters are updated during training. If the distribution, which a hidden layer of a CNN is trying to model, keeps on changing (i.e., the internal covariance shift is high), the training process will slow down and the network will take a long time to converge (simply because it is hard to reach a static target than to reach a continuously shifting target). The normalization of this distribution leads us to a consistent activation distribution during the training process, which enhances the convergence and avoids network instability issues such as the vanishing/exploding gradients and activation saturation.

Reflecting on what we have already studied in Chapter 4, this normalization step is similar to the whitening transform (applied as an input pre-processing step) which enforces the inputs to follow a unit Gaussian distribution with zero mean and unit variance. However, different to the whitening transform, batch normalization is applied to the intermediate CNN activations and can be integrated in an end-to-end network because of its differentiable computations.

The batch normalization operation can be implemented as a layer in a CNN. Given a set of activations $\{\mathbf{x}^i : i \in [1, m]\}$ (where $\mathbf{x}^i = \{x_j^i : j \in [1, n]\}$ has n dimensions) from a CNN layer corresponding to a specific input batch with m images, we can compute the first and second order statistics (mean and variance respectively) of the batch for each dimension of activations as follows:

$$\mu_{x_j} = \frac{1}{m} \sum_{i=1}^{m} x_j^i \tag{5.8}$$

$$\sigma_{x_j}^2 = \frac{1}{m} \sum_{i=1}^{m} (x_j^i - \mu_{x_j})^2 \tag{5.9}$$

μ_{x_j} and $\sigma_{x_j}^2$ represent the mean and variance for the j^{th} activation dimension computed over a batch, respectively. The normalized activation operation is represented as:

$$\hat{x}_j^i = \frac{x_j^i - \mu_{x_j}}{\sqrt{\sigma_{x_j}^2 + \epsilon}}. \tag{5.10}$$

Just the normalization of the activations is not sufficient, because it can alter the activations and disrupt the useful patterns that are learned by the network. Therefore, the normalized activations are rescaled and shifted to allow them to learn useful discriminative representations:

$$y_j^i = \gamma_j \hat{x}_j^i + \beta_j, \tag{5.11}$$

where γ_j and β_j are the learnable parameters which are tuned during error back-propagation.

Note that batch normalization is usually applied after the CNN weight layers, before applying the non-linear activation function. Batch normalization is an important tool that is used in state of the art CNN architectures (examples in Chapter 6). We briefly summarize the benefits of using batch normalization below:

- In practice, the network training becomes less sensitive to hyper-parameter choices (e.g., learning rate) when batch normalization is used [Ioffe and Szegedy, 2015].
- It stabilizes the training of very deep networks and provides robustness against bad weight initializations. It also avoids the vanishing gradient problem and the saturation of activation functions (e.g., tanh and sigmoid).
- Batch normalization greatly improves the network convergence rate. This is very important because very deep network architectures can take several days (even with reasonable hardware resources) to train on large-scale datasets.
- It integrates the normalization in the network by allowing back-propagation of errors through the normalization layer, and therefore allows end-to-end training of deep networks.
- It makes the model less dependent on regularization techniques such as dropout. Therefore, recent architectures do not use dropout when batch normalization is extensively used as a regularization mechanism [He et al., 2016a].

5.2.5 ENSEMBLE MODEL AVERAGING

The ensemble averaging approach is another simple, but effective, technique where a number of models are learned instead of just a single model. Each model has different parameters due to different random initializations, different hyper-parameter choices (e.g., architecture, learning rate) and/or different sets of training inputs. The output from these multiple models is then combined to generate a final prediction score. The prediction combination approach can be a simple output averaging, a majority voting scheme or a weighted combination of all predictions. The final prediction is more accurate and less prone to over-fitting compared to each individual model in the ensemble. The committee of experts (ensemble) acts as an effective regularization mechanism which enhances the generalization power of the overall system.

5.2.6 THE ℓ^2 REGULARIZATION

The ℓ^2 regularization penalizes large values of the parameters $\mathbf{w}$ during the network training. This is achieved by adding a term with ℓ^2 norm of the parameter values weighted by a hyper-parameter λ, which decides on the strength of penalization (in practice half of the squared magnitude times λ is added to the error function to ensure a simpler derivative

term) . Effectively, this regularization encourages small and spread-out weight distributions over large values concentrated over only few neurons. Consider a simple network with only a single hidden layer with parameters $\mathbf{w}$ and output p_n, $n \in [1, N]$ when the output layer has N neurons. If the desired output is denoted by y_n, we can use an euclidean objective function with ℓ^2 regularization to update the parameters as follows:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{m=1}^{M} \sum_{n=1}^{N} (p_n - y_n)^2 + \lambda \parallel \mathbf{w} \parallel_2, \tag{5.12}$$

where, M denote the number of training examples. Note that, as we will discuss later, ℓ^2 regularization performs the same operation as the weight decay technique. This approach is called '*weight decay*' because applying ℓ^2 regularization means that the weights are updated linearly (since the derivative of the regularizer term is λw for each neuron).

5.2.7 THE ℓ^1 REGULARIZATION

The ℓ^1 regularization technique is very similar to the ℓ^2 regularization, with the only difference being that the regularizer term uses the ℓ^1 norm of weights instead of an ℓ^2 norm. A hyper-parameter λ is used to define the strength of regularization. For a single layered network with parameters $\mathbf{w}$, we can denote the parameter optimization process using ℓ_1 norm as follows:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{m=1}^{M} \sum_{n=1}^{N} (p_n - y_n)^2 + \lambda \parallel \mathbf{w} \parallel_1, \tag{5.13}$$

where, N and M denote the number of output neurons and the number of training examples respectively. This effectively leads to sparse weight vectors for each neuron with most of the incoming connections having very small weights.

5.2.8 ELASTIC NET REGULARIZATION

Elastic net regularization linearly combines both ℓ^1 and ℓ^2 regularization techniques by adding a term $\lambda_1|w| + \lambda_2 w^2$ for each weight value. This results in sparse weights and often performs better than the individual ℓ_1 and ℓ_2 regularizations, each of which is a special case of elastic net regularization. For a single layered network with parameters $\mathbf{w}$, we can denote the parameter optimization process as follows:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{m=1}^{M} \sum_{n=1}^{N} (p_n - y_n)^2 + \lambda_1 \parallel \mathbf{w} \parallel_1 + \lambda_2 \parallel \mathbf{w} \parallel_2, \tag{5.14}$$

where, N and M denote the number of output neurons and the number of training examples respectively.

5.2.9 MAX-NORM CONSTRAINTS

The max-norm constraint is a form of regularization which puts an upper bound on the norm of the incoming weights of each neuron in a neural network layer. As a result, the weight vector $\mathbf{w}$ must follow the constraint $\| \mathbf{w} \|_2 < h$, where h is a hyper-parameter, whose value is usually set based on the performance of the network on a validation set. The benefit of using such a regularization is that the network parameters are guaranteed to remain in a reasonable numerical range even when high values of learning rates are used during network training. In practice, this leads to a better stability and performance [Srivastava et al., 2014].

5.2.10 EARLY STOPPING

The over-fitting problem occurs when a model performs very well on the training set but behaves poorly on unseen data. Early stopping is applied to avoid overfitting in the iterative gradient based algorithms. This is achieved by evaluating the performance on a held-out validation set at different iterations during the training process. The training algorithm can continue to improve on the training set until the performance on the validation set also improves. Once there is a drop in the generalization ability of the learned model, the learning process can be stopped or slowed down (Figure 5.2).

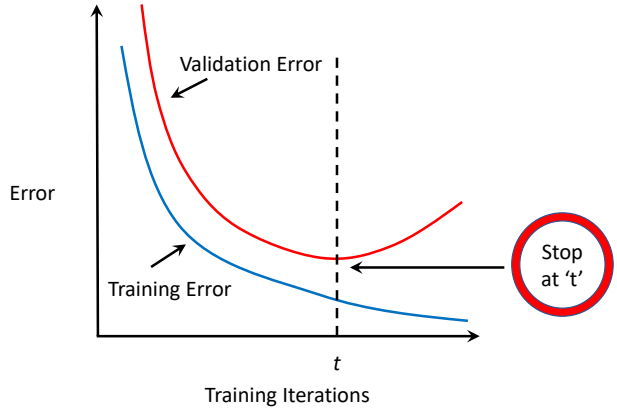

Figure 5.2: An illustration of the early stopping approach during network training.

Having discussed the concepts which enable successful training of deep neural networks (e.g., correct weight initialization and regularization techniques), we dive into the details of the network learning process. Gradient based algorithms are the most important tool to optimally train such networks on large-scale datasets. In the following, we discuss different variants of optimizers for CNNs.

5.3 GRADIENT BASED CNN LEARNING

The CNN learning process tunes the parameters of the network such that the input space is correctly mapped to the output space. As discussed before, at each trainig step, the current estimate of the output variables is matched with the desired output (often termed the '*ground-truth*' or the '*label space*'). This matching function serves as an objective function during the CNN training and it is usually called the loss function or the error function. In other words, we can say that the CNN training process involves the optimization of its parameters such that the **loss function** is minimized. The CNN parameters are the free/tunable weights in each of its layers (e.g., filter weights and biases of the convolution layers) (Chapter 4).

An intuitive, but simple way to approach this optimization problem is by repeatedly updating the parameters such that the loss function **progressively** reduces to a minimum value. It is important to note here that the optimization of non-linear models (such as CNNs) is a hard task, exacerbated by the fact that these models are mostly composed of a large number of tunable parameters. Therefore instead of solving for a globally optimal solution, we iteratively search for the locally optimal solution at each step. Here, the gradient based methods come as a natural choice, since we need to update the parameters in the **direction** of the steepest descent. The amount of parameter update, or the size of the update step is called the '**learning rate**'. Each iteration which updates the parameters using the complete training set is called a '**training epoch**'. We can write each training iteration at time t using the following parameter update equation:

$$\theta_t = \theta_{t-1} - \eta\, \delta_t \tag{5.15}$$

$$s.t., \quad \delta_t = \nabla_\theta \mathcal{F}(\theta_t), \tag{5.16}$$

where $\mathcal{F}(\cdot)$ denotes the function represented by the neural network with parameters θ, ∇ represents the gradient and η denotes the learning rate.

5.3.1 BATCH GRADIENT DESCENT

As we discussed in the previous section, gradient descent algorithms work by computing the gradient of the objective function with respect to the neural network parameters, followed by a parameter update in the direction of the steepest descent. The basic version of the gradient descent, termed '*batch gradient descent*', computes this gradient on the entire training set. It is guaranteed to converge to the global minimum for the case of convex problems. For non-convex problems, it can still attain a local minimum. However, the training sets can be very large in computer vision problems, and therefore learning via the batch gradient descent can be prohibitively slow because for each parameter update, it needs to compute the gradient on the complete training set. This leads us to the stochastic gradient descent, which effectively circumvents this problem.

5.3.2 STOCHASTIC GRADIENT DESCENT

Stochastic Gradient Descent (SGD) performs a parameter update for each set of input and output that are present in the training set. As a result it converges much faster compared to the batch gradient descent. Furthermore, it is able to learn in an '*online manner*', where the parameters can be tuned in the presence of new training examples. The only problem is that its convergence behavior is usually unstable, especially for relatively larger learning rates and when the training datasets contain diverse examples. When the learning rate is appropriately set, the SGD generally achieves a similar convergence behavior, compared to the batch gradient descent, for both the convex and non-convex problems.

5.3.3 MINI-BATCH GRADIENT DESCENT

Finally, the mini-batch gradient descent method is an improved form of the stochastic gradient descent approach, which provides a decent trade-off between convergence efficiency and convergence stability by dividing the training set into a number of mini-batches, each consisting of a relatively small number of training examples. The parameter update is then performed after computing the gradients on each mini-batch. Note that the training examples are usually randomly shuffled to improve homogeneity of the training set. This ensures a better convergence rate compared to the Batch Gradient Descent and a better stability compared to the Stochastic Gradient Descent [Ruder, 2016].

5.4 NEURAL NETWORK OPTIMIZERS

After a general overview of the gradient descent algorithms in Section 5.3, we can note that there are certain caveats which must be avoided during the network learning process. As an example, setting the learning rate can be a tricky endeavor in many practical problems. The training process is often highly affected by the parameter initialization. Furthermore, the vanishing and exploding gradients problems can occur especially for the case of deep networks. The training process is also susceptible to get trapped into a local minima, saddle points or a high error plateau where the gradient is approximately zero in every direction [Pascanu et al., 2014]. Note that the saddle points (also called the '*minmax points*') are those stationary points on the surface of the function, where the partial derivative with respect to its dimensions becomes zero (Figure 5.3). In the following discussion we outline different methods to address these limitations of the gradient descent algorithms. Since our goal is to optimize over high-dimensional parameter spaces, we will restrict our discussion to the more feasible first-order methods and will not deal with the high-order methods (Newton's method) which are ill-suited for large datasets.

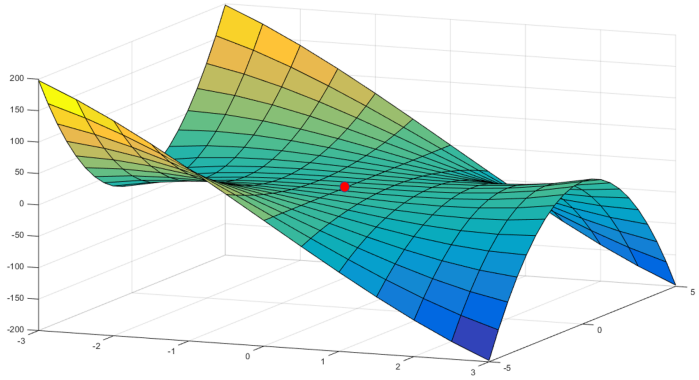

Figure 5.3: A saddle point shown as a red dot on a 3D surface. Note that the gradient is effectively zero, but it corresponds to neither a 'minima' nor a 'maxima' of the function.

5.4.1 MOMENTUM

Momentum based optimization provides an improved version of SGD with better convergence properties. For example, the SGD can oscillate close to a local minima, resulting in an unnecessarily delayed convergence. The momentum adds the gradient calculated at the previous time-step (a_{t-1}) weighted by a parameter γ to the weight update equation as follows:

$$\theta_t = \theta_{t-1} - a_t \tag{5.17}$$
$$a_t = \eta \nabla_\theta \mathcal{F}(\theta_t) + \gamma a_{t-1} \tag{5.18}$$

where $\mathcal{F}(\cdot)$ denotes the function represented by the neural network with parameters θ, ∇ represents the gradient and η denotes the learning rate.

The momentum term has physical meanings. The dimensions whose gradients point in the same direction are magnified quickly, while those dimensions whose gradients keep on changing directions are suppressed. Essentially, the convergence speed is increased because unnecessary oscillations are avoided. This can be understood as adding more momentum to the ball so that it moves along the direction of the maximum slope. Typically, the momentum is set to 0.9 during the SGD based learning.

5.4.2 NESTEROV MOMENTUM

The momentum term introduced in the previous section would carry the ball beyond the minimum point. Ideally, we would like the ball to slow down when the ball reaches the minimum point and the slope starts ascending. This is achieved by the Nesterov momentum

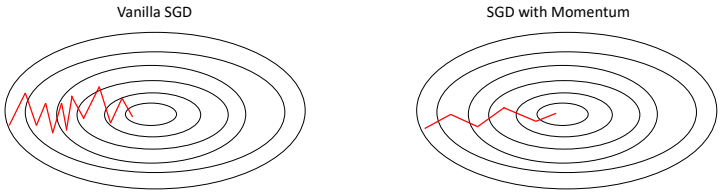

Figure 5.4: Comparison of the convergence behavior of the SGD with (*right*) and without (*left*) momentum.

[Nesterov, 1983] which computes the gradient at the next approximate point during the parameter update process, instead of the current point. This gives the algorithm the ability to 'look ahead' at each iteration and plan a jump such that the learning process avoids uphill steps. The update process can be represented as:

$$\theta_t = \theta_{t-1} - a_t \tag{5.19}$$
$$a_t = \eta \nabla_\theta \mathcal{F}(\theta_t - \gamma a_{t-1}) + \gamma a_{t-1} \tag{5.20}$$

where $\mathcal{F}(\cdot)$ denotes the function represented by the neural network with parameters θ, ∇ represents the gradient operation, η denotes the learning rate and γ is the momentum.

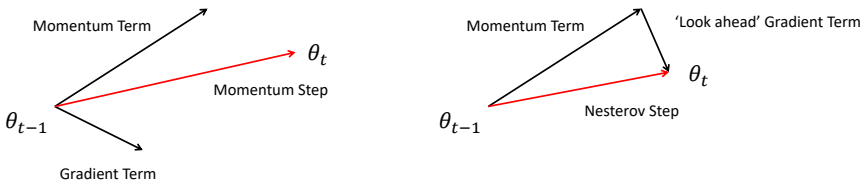

Figure 5.5: Comparison of the convergence behavior of the SGD with momentum (*left*) and the Nesterov update (*right*). While momentum update can carry the solver quickly towards a local optima, it can overshoot and miss the optimum point. A solver with Nesterov update corrects its update by looking ahead and correcting for the next gradient value.

5.4.3 ADAPTIVE GRADIENT

The momentum in the SGD refines the update direction along the slope of the error function. However, all parameters are updated at the same rate. In several cases, it is more useful to update each parameter differently, depending on its frequency in the training set or its significance for our end problem.

Adaptive Gradient (AdaGrad) algorithm [Duchi et al., 2011] provides a solution to this problem by using an adaptive learning rate for each individual parameter i. This is done at each time step t by dividing the learning rate of each parameter with the accumulation of the square of all the historical gradients for each parameter θ_i. This can be shown as follows:

$$\theta_t^i = \theta_{t-1}^i - \frac{\eta}{\sqrt{\sum_{\tau=1}^{t} {\delta_\tau^i}^2 + \epsilon}} \delta_t^i \tag{5.21}$$

where, δ_t^i is the gradient at time-step t with respect to the parameter θ_i and ϵ is a very small term in the denominator to avoid division by zero. The adaptation of the learning rate for each parameter removes the need to manually set the value of the learning rate. Typically, η is kept fixed to a single value (e.g., 10^{-2} or 10^{-3}) during the training phase. Note that AdaGrad works very well for sparse gradients, for which a reliable estimate of the past gradients is obtained by accumulating all of the previous time steps.

5.4.4 ADAPTIVE DELTA

Although AdaGrad eliminates the need to manually set the value of the learning rate at different epochs, it suffers from the vanishing learning rate problem. Specifically, as the number of iterations grows (t is large), the sum of the squared gradients becomes large, making the effective learning rate very small. As a result, the parameters do not change in the subsequent training iterations. Lastly, it also needs an initial learning rate to be set during the training phase.

The Adaptive Delta (AdaDelta) algorithm [Zeiler, 2012] solves both these problems by accumulating only the last k gradients in the denominator term of Equation 5.21. Therefore, the new update step can be represented as follows:

$$\theta_t^i = \theta_{t-1}^i - \frac{\eta}{\sqrt{\sum_{\tau=t-k+1}^{t} {\delta_\tau^i}^2 + \epsilon}} \delta_t^i. \tag{5.22}$$

This requires the storage of the last k gradients at each iteration. In practice, it is much easier to work with a running average $E[\delta^2]_t$, which can be defined as:

$$E[\delta^2]_t = \gamma E[\delta^2]_{t-1} + (1-\gamma){\delta_t}^2. \tag{5.23}$$

Here, γ has a similar function to the momentum parameter. Note that the above function implements an exponentially decaying average of the squared gradients for each parameter.

The new update step is:

$$\theta_t^i = \theta_{t-1}^i - \frac{\eta}{\sqrt{E[\delta^2]_t + \epsilon}}\delta_t^i, \tag{5.24}$$

$$\Delta\theta = -\frac{\eta}{\sqrt{E[\delta^2]_t + \epsilon}}\delta_t^i. \tag{5.25}$$

Note that we still did not get rid of the initial learning rate η. Zeiler [Zeiler, 2012] noted that this can be avoided by making the units of the update step consistent by introducing a Hessian approximation in the update rule. This boils down to the following:

$$\theta_t^i = \theta_{t-1}^i - \frac{\sqrt{E[(\Delta\theta)^2]_{t-1} + \epsilon}}{\sqrt{E[\delta^2]_t + \epsilon}}\delta_t^i. \tag{5.26}$$

Note that we have considered here the local curvature of the function $\mathcal{F}$ to be approximately flat and replaced $E[(\Delta\theta)^2]_t$ (not known) with $E[(\Delta\theta)^2]_{t-1}$ (known).

5.4.5 RMSPROP

RMSprop [Tieleman and Hinton, 2012] is closely related to the AdaDelta approach, aiming to resolve the vanishing learning rate problem of AdaGrad. Similar to AdaDelta, it also calculates the running average as follows:

$$E[\delta^2]_t = \gamma E[\delta^2]_{t-1} + (1-\gamma){\delta_t}^2. \tag{5.27}$$

Here, a typical value of γ is 0.9. The update rule of the tunable parameters takes the following form:

$$\theta_t^i = \theta_{t-1}^i - \frac{\eta}{\sqrt{E[\delta^2]_t + \epsilon}}\delta_t^i. \tag{5.28}$$

5.4.6 ADAPTIVE MOMENT ESTIMATION

We stated that the AdaGrad solver suffers from the vanishing learning rate problem, but AdaGrad is very useful for cases where gradients are sparse. On the other hand, RMSprop does not reduce the learning rate to a very small value at higher time steps. However on the negative side, it does not provide an optimal solution for the case of sparse gradients. The ADAptive Moment Estimation (ADAM) [Kingma and Ba, 2014] approach estimates a separate learning rate for each parameter and combines the positives of both AdaGrad and RMSprop. The main difference between Adam and its two predecessors (RMSprop and AdaDelta) is that the updates are estimated by using both the first moment and the second moment of the gradient (as in Equations 5.26 and 5.28). Therefore a running average of gradients (mean) is maintained along with a running average of the squared gradients

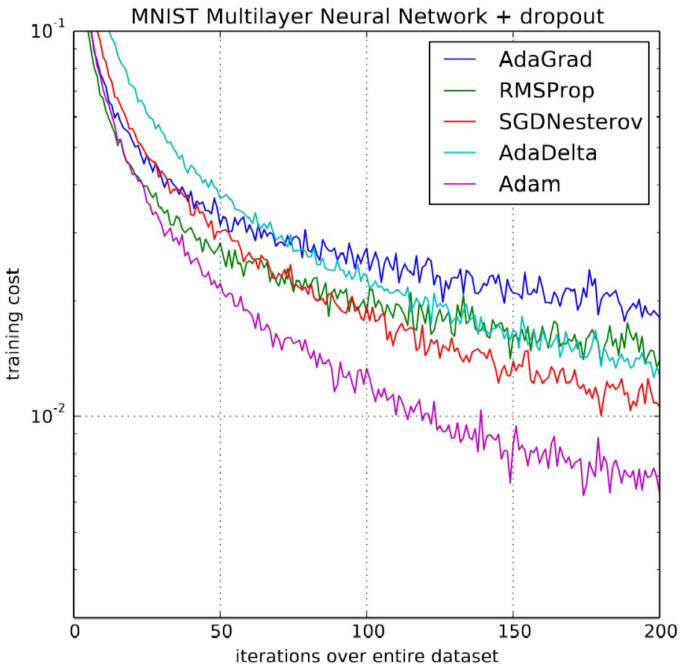

Figure 5.6: Convergence performance on the MNIST dataset using different neural network optimizers [Kingma and Ba, 2014] .

(variance) as follows:

$$E[\delta]_t = \gamma_1 E[\delta]_{t-1} + (1-\gamma_1)\delta_t, \tag{5.29}$$
$$E[\delta^2]_t = \gamma_2 E[\delta^2]_{t-1} + (1-\gamma_2){\delta_t}^2, \tag{5.30}$$

where, γ_1 and γ_2 are the parameters for running averages of the mean and the variance respectively. Since the initial moment estimates are set to zero, they can remain very small even after many iterations, especially when $\gamma_{1,2} \neq 1$. To overcome this issue, the initialization bias-corrected estimates of $E[\delta]_t$ and $E[\delta^2]_t$ are obtained as follows:

$$\hat{E}[\delta]_t = \frac{E[\delta]_t}{1-(\gamma_1)^t} \tag{5.31}$$
$$\hat{E}[\delta^2]_t = \frac{E[\delta^2]_t}{1-(\gamma_2)^t} \tag{5.32}$$

Very similar to what we studied in the case of AdaGrad, AdaDelta and RMSprop, the update rule for Adam is given by:

$$\theta_t^i = \theta_{t-1}^i - \frac{\eta}{\sqrt{\hat{E}[\delta^2]_t + \epsilon}} \hat{E}[\delta]_t. \tag{5.33}$$

The authors found $\gamma_1 = 0.9, \gamma_2 = 0.999, \eta = 0.001$ to be good default values of the decay (γ) and learning (η) rates during the training process.

Figure 5.6 [Kingma and Ba, 2014] illustrates the convergence performance of the discussed solvers on the MNIST dataset for handwritten digit classification. Note that the SGD with Nesterov shows a good convergence behavior, however it requires a manual tuning of the learning rate hyper-parameter. Among the solvers with an adaptive learning rate, Adam performs the best in this example (also beating the manually tuned SGD-Nesterov solver). In practice, Adam usually scales very well to large scale problems and exhibits nice convergence properties. That is why, Adam is often a default choice for many computer vision applications based on deep learning.

5.5 GRADIENT COMPUTATION IN CNNS

We have discussed a number of layers and architectures for CNNs. In Sec. 3.2.2, we also described the back-propagation algorithm used to train CNNs. In essence, back-propagation lies at the heart of CNN training. Error back-propagation can only happen if the CNN layers implement a differentiable operation. Therefore, it is interesting to study how the gradient can be computed for the different CNN layers. In this section, we will discuss in detail the different approaches which are used to compute the differentials of popular CNN layers.

We describe in the following the four different approaches which can be used to compute gradients.

5.5.1 ANALYTICAL DIFFERENTIATION

It involves the manual derivation of the derivatives of a function performed by a CNN layer. These derivatives are then implemented in a computer program to calculate the gradients. The gradient formulas are then used by an optimization algorithm (e.g., Stochastic Gradient Descent) to learn the optimal CNN weights.

Example: Assume for a simple function, $y = f(x) = x^2$, we want to calculate the derivative analytically. By applying the differentiation formula for polynomial functions, we can find the derivative as follows:

$$\frac{dy}{dx} = 2x, \tag{5.34}$$

which can give us the slope at any point x.

Analytically deriving the derivatives of complex expressions is time-consuming and laborious. Furthermore, it is necessary to model the layer operation as a closed-form mathematical expression. However, it provides an accurate value for the derivative at each point.

5.5.2 NUMERICAL DIFFERENTIATION

Numerical differentiation techniques use the values of a function to estimate the numerical value of the derivative of the function at a specific point.

Example: For a given function $f(x)$, we can estimate the first-order numerical derivative at a point x by using the function values at two nearby points i.e., $f(x)$ and $f(x+h)$, where h is a small change in x:

$$\frac{f(x+h)-f(x)}{h} \tag{5.35}$$

The above equation estimates the first-order derivative as the slope of a line joining the two points $f(x)$ and $f(x+h)$. The above expression is called the 'Newton's Difference Formula'.

Numerical differentiation is useful in cases where we know little about the underline real function or when the actual function is too complex. Also in several cases, we only have access to discrete sampled data (e.g., at different time instances) and a natural choice is to estimate the derivatives without necessarily modeling the function and calculating the exact derivatives. Numerical differentiation is fairly easy to implement, compared to other approaches. However, numerical differentiation provides only an estimate of the derivative and works poorly, particularly for the calculation of higher order derivatives.

5.5.3 SYMBOLIC DIFFERENTIATION

Symbolic differentiation uses standard differential calculus formulas to manipulate mathematical expressions using computer algorithms. Popular softwares which perform symbolic differentiation include Mathematica, Maple and Matlab.

Example: Suppose we are given a function $f(x) = \exp(\sin(x))$, we need to calculate its 10^{th} derivative with respect to x. An analytical solution would be cumbersome, while a numerical solution will be less accurate. In such cases, we can effectively use symbolic differentiation to get a

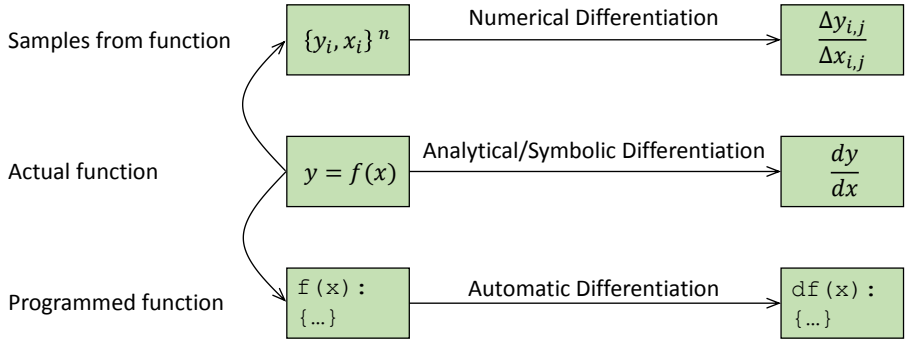

Figure 5.7: Relationships between different differentiation methods.

reliable answer. The following code in Matlab (using the Symbolic Math Toolbox) gives the desired result.

```
>> syms x
>> f(x) = exp(sin(x))
>> diff(f,x,10)
256*exp(sin(x))*cos(x)^2 - exp(sin(x))*sin(x) -
5440*exp(sin(x))*cos(x)^4 + 2352*exp(sin(x))*cos(x)^6 - ...
```

Symbolic differentiation, in a sense, is similar to analytical differentiation, but leveraging the power of computers to perform laborious derivations. This approach reduces the need to manually derive differentials and avoids the inaccuracies of numerical methods. However, symbolic differentiation often leads to complex and long expressions which results in slow software programs. Also, it does not scale well to higher order derivatives (similar to numerical differentiation) due to the high complexity of the required computations. Furthermore, in neural network optimization, we need to calculate partial derivatives with respect to a large number of inputs to a layer. In such cases, symbolic differentiation is inefficient and does not scale well to large-scale networks.

5.5.4 AUTOMATIC DIFFERENTIATION

Automatic differentiation is a powerful technique which uses both numerical and symbolic techniques to estimate the differential calculation in the software domain i.e., given a coded computer program which implements a function, automatic differentiation can be used to design another program which implements the derivative of that function. We illustrate the automatic differentiation and its relationship with the numerical and symbolic differentiation in Figure 5.7.

Algorithm 1 Forward mode of Automatic Differentiation

Input : $x, \mathcal{C}$

Output : y_n

1: $y_0 \leftarrow x$ % initialization
2: **for all** $i \in [1, n]$ **do**
3: $y_i \leftarrow f_i^e(y_{\mathrm{Pa}(f_i^e)})$ % each function operates on its parents output in the graph
4: **end for**

Every computer program is implemented using a programming language, which only supports a set of basic functions (e.g., addition, multiplication, exponentiation, logarithm and trigonometric functions). Automatic differentiation uses this modular nature of computer programs to break them into simpler elementary functions. The derivatives of these simple functions are computed symbolically and the chain rule is then applied repeatedly to compute any order of derivatives of complex programs.

Automatic differentiation provides an accurate and efficient solution to the differentiation of complex expressions. Precisely, automatic differentiation gives results which are accurate to the machine precision. The computational complexity of calculating derivatives of a function is almost the same as evaluating the original function itself. Unlike symbolic differentiation, it neither needs a closed form expression of the function, nor does it suffer from expression swell which renders symbolic differentiation inefficient and difficult to code. Current state of the art CNN libraries such as Theano and Tensorflow use automatic differentiation to compute derivatives (see Chapter 8).

Automatic differentiation is very closely related to the back-propagation algorithm we studied before in Section 3.2.2. It operates in two modes, the forward mode and the backward mode. Given a complex function, we first decompose it into a computational graph consisting of simple elementary functions which are joined with each other to compute the complex function. In the **forward** mode, given an input x, the computational graph $\mathcal{C}$ with n intermediate states (corresponding to $\{f^e\}^n$ elementary functions) can be evaluated sequentially as shown in Algorithm 1.

After the forward computations shown in Algorithm 1, the **backward** mode starts computing the derivatives towards the end and successively applies the chain rule to calculate the differential with respect to each intermediate output variable y_i as shown in Algorithm 2.

A basic assumption in the automatic differentiation approach is that the expression is differentiable. If this is not the case, automatic differentiation will fail. We provide a simple example of forward and backward modes of automatic differentiation below and refer the reader to Baydin et al. [2015] for a detailed treatment of this subject in relation to machine/deep learning techniques.

Algorithm 2 Backward mode of Automatic Differentiation

Input : x, $\mathcal{C}$

Output : $\frac{dy_n}{dx}$

1: Perform forward mode propagation
2: **for all** $i \in [n-1, 0]$ **do**
3: % chain rule to compute derivatives using child nodes in the graph
4: $\frac{dy_n}{dy_i} \leftarrow \sum_{j \in \text{Ch}(f_i^e)} \frac{dy_n}{dy_j} \frac{df_j^e}{dy_i}$
5: **end for**
6: $\frac{dy_n}{dx} \leftarrow \frac{dy_n}{dy_0}$

Example: Consider a slightly more complex function than the previous example for symbolic differentiation,

$$y = f(x) = \exp(\sin(x) + \sin(x)^2) + \sin(\exp(x) + \exp(x)^2). \tag{5.36}$$

We can represent its analytically or symbolically calculated differential as follows:

$$\begin{aligned} \frac{df}{dx} &= \cos(\exp(2x) + \exp(x))(2\exp(2x) + \exp(x)) \\ &\quad + \exp(\sin(x)^2 + \sin(x))(\cos(x) + 2\cos(x)\sin(x)) \end{aligned} \tag{5.37}$$

But, if we are interested in calculating its derivative using automatic differentiation, the first step would be to represent the complete function in terms of basic operations (addition, exp and sin) defined as:

$$\begin{array}{lll} a = \sin(x) & b = a^2 & c = a + b \\ d = \exp(c) & e = \exp(x) & f = e^2 \\ g = e + f & h = \sin(g) & y = d + h \end{array} \tag{5.38}$$

The flow of computations in terms of these basic operations is illustrated in the computational graph in Fig. 5.8. Given this computational graph we can easily calculate the differential of the output with respect to each

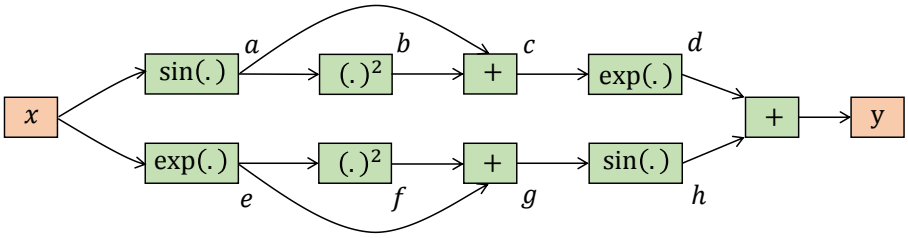

Figure 5.8: Computational graph showing the calculation of our desired function.

of the variables in the graph as follows:

$$\frac{dy}{dd} = 1 \qquad \frac{dy}{dh} = 1 \qquad \frac{dy}{dc} = \frac{dy}{dd}\frac{dd}{dc} \qquad \frac{dy}{dg} = \frac{dy}{dh}\frac{dh}{dg}$$
$$\frac{dy}{db} = \frac{dy}{dc}\frac{dc}{db} \qquad \frac{dy}{da} = \frac{dy}{dc}\frac{dc}{da} + \frac{dy}{db}\frac{db}{da} \qquad \frac{dy}{df} = \frac{dy}{dg}\frac{dg}{df}$$
$$\frac{dy}{de} = \frac{dy}{dg}\frac{dg}{de} + \frac{dy}{df}\frac{df}{de} \qquad \frac{dy}{dx} = \frac{dy}{da}\frac{da}{dx} + \frac{dy}{de}\frac{de}{dx} \tag{5.39}$$

All of the above differentials can easily be computed because it is simple to compute the derivative of each basic function, e.g.:

$$\frac{dd}{dc} = \exp(c) \qquad \frac{dh}{dg} = \cos(g) \qquad \frac{dc}{db} = \frac{dc}{da} = 1 \qquad \frac{df}{de} = 2e \tag{5.40}$$

Note that we started towards the end of the computational graph, and computed all of the intermediate differentials moving backwards, until we got the differential with respect to input. The original differential expression we calculated in Equation 5.37 was quite complex. However, once we decomposed the original expression in to simpler functions in Equation 5.38, we note that the complexity of the operations that are required to calculate the derivative (back-ward pass) is almost the same as the calculation of the original expression (forward pass) according to the computational graph. Automatic differentiation uses the forward and backward operation modes to efficiently and precisely calculate the differentials of complex functions. As we discussed above, the operations for the calculation of differentials in this manner have a close resemblance to the back-propagation algorithm.

We have completed the discussion about the CNN architecture and its training process. Next, we will describe a number of successful CNN examples from the literature which will help us develop an insight into the state-of-the-art network topologies and their mutual pros and cons.