Neural Net and Deep Learning

by Xia, Tingfeng

Wednesday 12th February, 2020

This work is licensed under a Creative Commons "Attribution-NonCommercial-ShareAlike 4.0 International" license.

Contents

1 Optimization **3**
1.1 Prerequisites . . . 3
1.2 Determining Curvature . . . 3
1.2.1 Definite-ness . . . 4
1.3 Convexity . . . 4
1.3.1 Convex Sets . . . 4
1.3.2 Convex Functions . . . 4
1.3.3 Convexity Characterization with Hessian . . . 5
1.3.4 Convexity Composition Properties . . . 5
1.3.5 Convexity for Linear Models . . . 5
1.3.6 Non-Convexity of Networks . . . 5
1.4 Nuisance-ness of Cost Surface . . . 6
1.4.1 Saddle Point . . . 6
1.4.2 Plateaux . . . 7
1.4.3 Ill-Conditioned Curvature . . . 7
1.5 Alternative Gradient Descent . . . 9
1.5.1 Momentum Gradient Descent . . . 9
1.5.2 Stochastic Gradient Descent . . . 10
1.5.3 Minibatch SGD . . . 10
1.5.4 SGD Learning Rate . . . 11
1.6 Debugging Optimization . . . 12

2 Convolutional Neural Net and Image Classification **12**
2.1 Motivations for Convolution Layer 12
2.2 The Convolution Operator 12
2.2.1 1-D Signal Processing 12
2.2.2 2-D Convolution 13
2.2.3 Properties of Convolution 13
2.3 Canonical Kernels 13
2.3.1 Blurring Kernel 13
2.3.2 Sharpening Kernel 13
2.3.3 Edge Detector Kernel 13
2.3.4 Vertical Edge Detector Kernel 14
2.4 Convolutional Networks 14
2.4.1 Pooling Layer 14
2.4.2 Non-linearity in Convolutional Layers 14
2.4.3 Equivariance and Invariance 14
2.4.4 Channels in Convolution Layers 15
2.5 Size of Convolutional Neural Nets 15
2.5.1 Rule of thumb on size 16
2.6 Supervised Pre-training and Transfer Learning 16

References **18**

1 Optimization

1.1 Prerequisites

Clairaut's Theorem states that second order derivatives are such that

$$\frac{\partial^2 \mathcal{J}}{\partial \theta_i \partial \theta_j} = \frac{\partial^2 \mathcal{J}}{\partial \theta_j \partial \theta_i} \tag{1.1}$$

The Hessian Matrix is a symmetric matrix (due to Clairaut's Theorem) defined as

$$\mathbf{H} = \nabla^2 \mathcal{J} = \begin{pmatrix} \frac{\partial^2 \mathcal{J}}{\partial \theta_1^2} & \frac{\partial^2 \mathcal{J}}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 \mathcal{J}}{\partial \theta_1 \partial_D} \\ \frac{\partial^2 \mathcal{J}}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 \mathcal{J}}{\partial \theta_2^2} & \cdots & \frac{\partial^2 \mathcal{J}}{\partial \theta_2 \partial \theta_D} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 \mathcal{J}}{\partial \theta_D \partial \theta_1} & \frac{\partial^2 \mathcal{J}}{\partial \theta_D \partial \theta_2} & \cdots & \frac{\partial^2 \mathcal{J}}{\partial \theta_D^2} \end{pmatrix} \tag{1.2}$$

Second-Order Taylor Approximation Locally, a function can be approximated by its second order Taylor approximation. We have, for $\boldsymbol{\theta}$ that is sufficiently close to $\boldsymbol{\theta}_0$

$$\mathcal{J}(\boldsymbol{\theta}) \approx \mathcal{J}(\boldsymbol{\theta}_0) + \underbrace{\nabla \mathcal{J}(\boldsymbol{\theta}_0)^\top (\boldsymbol{\theta} - \boldsymbol{\theta}_0)}_{1st\ order} + \underbrace{\frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H}(\boldsymbol{\theta}_0) (\boldsymbol{\theta} - \boldsymbol{\theta}_0)}_{quadratic} \tag{1.3}$$

Notice that if $\boldsymbol{\theta}$ is a critical point, then the gradient would be zero. In such case, the approximation will be

$$\mathcal{J}(\boldsymbol{\theta}) \approx \mathcal{J}(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H}(\boldsymbol{\theta}_0) (\boldsymbol{\theta} - \boldsymbol{\theta}_0) \tag{1.4}$$

Spectral Decomposition Since the Hessian matrix is symmetric, we know that it is spectral decompose-able, that is

$$\mathbf{H} = \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^\top = \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^{-1} \tag{1.5}$$

Notice that orthogonal matrices are ***necessarily*** invertible, with transpose and inverse equal to each other.

where $\mathbf{Q}$ is the (orthogonal) matrix where columns are the eigenvectors and $\boldsymbol{\Lambda}$ is the diagonal matrix with eigenvalues on diagonal.

1.2 Determining Curvature

Often, we refer to the hessian matrix $\mathbf{H}$ as the curvature of a function. Suppose that we are moving along a line defined by $\boldsymbol{\theta} + t\mathbf{v}$ for some vector $\mathbf{v}$. According to the second order Taylor Approximation, we have

$$\mathcal{J}(\boldsymbol{\theta} + t\mathbf{v}) \approx \mathcal{J}(\boldsymbol{\theta}) + t\nabla \mathcal{J}(\boldsymbol{\theta})^\top \mathbf{v} + \frac{t^2}{2} \mathbf{v}^\top \mathbf{H}(\boldsymbol{\theta}) \mathbf{v} \tag{1.6}$$

Thus, in the case where $\mathbf{v}^\top \mathbf{H}\mathbf{v} > 0$, the cost function curves up, i.e. has ***positive curvature*** where as in the case $\mathbf{v}^\top \mathbf{H}\mathbf{v} < 0$, it has a ***negative curvature***.

1.2.1 Definite-ness

- A matrix $\mathbf{A}$ is positive definite if $\mathbf{v}^\top \mathbf{A}\mathbf{v} > 0, \forall \mathbf{v} \neq \mathbf{0}$.
- We say it is positive semidefinite if $\mathbf{v}^\top \mathbf{A}\mathbf{v} \geq 0, \forall \mathbf{v} \neq \mathbf{0}$.
- Equivalently, matrix is positive definite *if and only if* all its eigenvalues are positive, and is positive semidefinite *iff* all its eigenvalues are non-negative. I will now show the case (one direction) for positive definite-ness.
  Proof: Let $\mathbf{v} \neq 0 \in \mathbb{R}^n$, then we know

$$\mathbf{v}^\top A\mathbf{v} = \mathbf{v}^\top Q\Lambda Q^\top \mathbf{v} = (\dagger) \tag{1.7}$$

  Now, if we define $\mathbf{z} = \mathbf{v}^\top Q$, we have

$$(\dagger) = \mathbf{z}\Lambda\mathbf{z}^\top = \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix} \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} \begin{bmatrix} z_1 & \dots & z_n \end{bmatrix} = \sum_i \lambda_i z_i^2 \tag{1.8}$$

  Since $\mathbf{v} \neq \mathbf{0}$, we know $\mathbf{z} = Q^\top \mathbf{v} \neq \mathbf{0}$, we know that $(\dagger)$ must be positive and this concludes the proof. ■

- For any critical point $\boldsymbol{\theta}_*$, if $\mathbf{H}(\boldsymbol{\theta}_*)$ exists and is positive definite, then $\boldsymbol{\theta}_*$ is a local minimum.

In naïve words, all directions curve upwards.

1.3 Convexity

1.3.1 Convex Sets

Recall that we say set $\mathcal{S}$ is convex if any $\mathbf{x}_0, \mathbf{x}_1 \in \mathcal{S}$, it holds that

$$(1-\lambda)\mathbf{x}_0 + \lambda\mathbf{x}_1 \in \mathcal{S} \quad \text{for } 0 \leq \lambda \leq 1 \tag{1.9}$$

1.3.2 Convex Functions

We say that a function f is convex if for any $\mathbf{x}_0, \mathbf{x}_1$ it holds that

$$f\left((1-\lambda)\mathbf{x}_0 + \lambda\mathbf{x}_1\right) \leq (1-\lambda)f\left(\mathbf{x}_0\right) + \lambda f\left(\mathbf{x}_1\right) \quad \text{for } 0 \leq \lambda \leq 1 \tag{1.10}$$

1.3.3 Convexity Characterization with Hessian

If $\mathcal{J}$ is twice differentiable, then we can use a equivalent characterization in terms of the Hessian matrix $\mathbf{H}$

- A twice diff-able function is convex *iff* its Hessian is positive semidefinite everywhere the function is defined.
- In the case of univariate function, where the Hessian would be essentially 1 by 1, we say it is convex *iff* its second derivative is non-negative everywhere.

Possibly add proof for convexity of squared error, logistic CE, and softmax CE.

1.3.4 Convexity Composition Properties

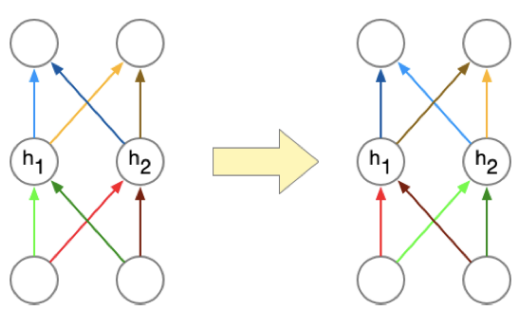
- The composition of two convex functions is convex.
- If f is a non-decreasing univariate function and g is a convex function, then $f \circ g$ is convex.

1.3.5 Convexity for Linear Models

For a linear model, have the hypothesis $z = \mathbf{w}^\top \mathbf{x} + b$ is a linear function of $\mathbf{w}$ and b. Now, if the loss is convex as a function of z, then it is convex as a function of $\mathbf{w}$ and b. Thus, linear regression, logistic regression, and softmax regression are clearly convex. (by directly applying the rules in the Convexity Composition Properties section above.)

1.3.6 Non-Convexity of Networks

Clearly, we want the cost function of interest to be convex, since if that is the case then there will be no *spurious local minima*, meaning that any local minimum will also be a global minimum. In such case, learning is very convenient since gradient based algorithms can just go downhill all the way. Unfortunately, training a network with hidden units cannot be convex due to permutation symmetries – we can re-permute the hidden units in a way that preserves the function computed by the network. An example from slides[1] gives a good intuition of this matter.



Multilayer Neural Net Non-Convex Argument

- By definition, if a (cost) function $\mathcal{J}$ is convex, then for any set of points $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_N$ in its domain, we know

$$\mathcal{J}\left(\lambda_1 \boldsymbol{\theta}_1 + \cdots + \lambda_N \boldsymbol{\theta}_N\right) \leq \lambda_1 \mathcal{J}\left(\boldsymbol{\theta}_1\right) + \cdots + \lambda_N \mathcal{J}\left(\boldsymbol{\theta}_N\right) \quad \text{for } \lambda_i \geq 0, \sum_i \lambda_i = 1 \quad (1.11)$$

- Due to permutation symmetry, there are $K!$ (where K is the number of hidden units in a given hidden layer) permutations of the hidden units in a given layer which all compute the same function.
- Suppose that we average the parameters for all $K!$ permutations, then we get a degenerate network where all the hidden units are identical (i.e. the network learnt nothing using these identical weights).
- Under the assumption that the cost function was convex, this solution would have to be better than the original one (i.e. the one before we average out all the weights in hidden units) and this leads to a contradiction.[1.1] —⋊—Thus training a multilayer neural nets is non-convex. ■

1.4 Nuisance-ness of Cost Surface

In this sub-section, we will discuss how non-nice

1.4.1 Saddle Point

A saddle point is a point if

- $\nabla \mathcal{J}(\boldsymbol{\theta}) = \mathbf{0}$, and
- $\mathbf{H}(\boldsymbol{\theta})$ has some positive and some negative eigenvalues, i.e. some directions with positive curvature and some with negative curvature.

This could be troublesome since

- If we are exactly on the saddle point, then we are stuck - gradient descent won't get us any further.
- In such case, if we inspect the cost, we would judge that the algorithm has converged. However the solution acquired could be far from optimal.

We can tackle this problem by perturbing the current weight solution slightly in the weight space so that the weight could start moving again. It is important that we don't initialize all of the weights to zero, we should break the symmetry by using small random values.

[1.1] Remark: Generally, it is safe to say that a network that has leant ***absolutely nothing*** would have a much higher cost than a network that, at least, converged to a local minimum

1.4.2 Plateaux

Plateaux happen more than we think they do, here are some examples that very common

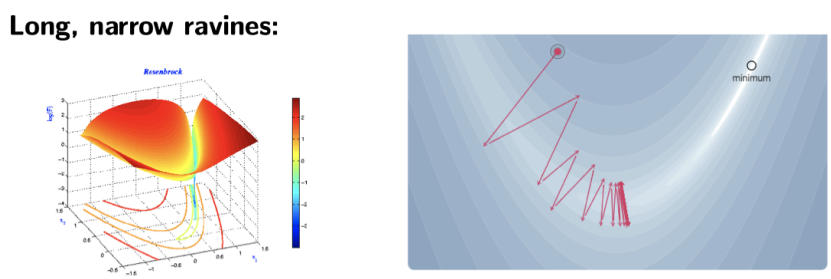
- 0-1 loss
- hard threshold activations
- logistic activations and least square

Saturated Unit is when we are in the flat region of the activation function. In the case of logistic non-linearity ($y = 1/(1 + e^{-x})$), when we have very large or very small x's the gradient would be very close to zero and thus the weights update will get stuck (or be extremely slow).

Dead Unit describes an even more extreme case where the gradient is exactly zero. For example, the negative half of the ReLU activation.

1.4.3 Ill-Conditioned Curvature

[1.2] Below is a typical example[1] of a Ill-conditioned curvature



As we can see from the gradient updates shown in the level set graph, the solution that we get at each iteration bounces back and forth in the high curvature directions and makes very minimal progress in the low curvature direction.

We can characterize such curvature by noticing that the hessian $\mathbf{H}$ would have some *large* positive eigenvalues and some eigenvalues close to zero, corresponding to high-curvature and low-curvature directions respectively.

[1.2] This is very common in neural net training.

Gradient Descent Dynamics Consider a quadratic objective (which is convex)

$$\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{\theta}^\top \boldsymbol{A}\boldsymbol{\theta} \qquad \text{where } \mathbf{A} \text{ is } +\text{-ive semi definite} \tag{1.12}$$

then the gradient descent updates would be

$$\begin{aligned}
\boldsymbol{\theta}_{k+1} &\leftarrow \boldsymbol{\theta}_k - \alpha \nabla \mathcal{J}(\boldsymbol{\theta}_k) && (1.13)\\
&= \boldsymbol{\theta}_k - \alpha \mathbf{A}\boldsymbol{\theta}_k && (1.14)\\
&= (\mathbf{I} - \alpha\mathbf{A})\boldsymbol{\theta}_k && (1.15)\\
\implies \boldsymbol{\theta}_k &= (\mathbf{I} - \alpha\mathbf{A})^k \boldsymbol{\theta}_0 && (1.16)
\end{aligned}$$

Let $\mathbf{A} = \mathbf{Q} \wedge \mathbf{Q}^\top$ be the spectral decomposition of $\mathbf{A}$, then

$$\begin{aligned}
(\mathbf{I} - \alpha\mathbf{A})^k \boldsymbol{\theta}_0 &= \left(\mathbf{I} - \alpha\mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^\top\right)^k \boldsymbol{\theta}_0 && (1.17)\\
&= \left[\mathbf{Q}(\mathbf{I} - \alpha\boldsymbol{\Lambda})\mathbf{Q}^\top\right]^k \boldsymbol{\theta}_0 && (1.18)\\
&= \mathbf{Q}(\mathbf{I} - \alpha\boldsymbol{\Lambda})^k \mathbf{Q}^\top \boldsymbol{\theta}_0 \qquad \mathbf{Q} \text{ is orthonormal, thus } \mathbf{Q}^\top\mathbf{Q} = \mathbf{I} && (1.19)\\
&= \mathbf{Q}(\mathbf{I} - \alpha\boldsymbol{\Lambda})^k \mathbf{Q}^{-1} \boldsymbol{\theta}_0 && (1.20)
\end{aligned}$$

In $\mathbf{Q}$ basis, each coordinate gets multiplied by $(1 - \alpha\lambda_i)^k$ where λ_i's denote eigenvalues of $\mathbf{A}$. Hence,

- If $0 < \alpha\lambda_i \leq 1$: decays to zero at a rate that depends on $\alpha\lambda_i$
- If $1 < \alpha\lambda_i \leq 2$: oscillations will happen
- If $\alpha\lambda_i > 2$: then the gradient updates diverges. (unstable)

The result is that we will need to set our learning rate such that

$$\alpha < \frac{2}{\lambda_{\max}} \qquad \text{where } \lambda_{\max} \triangleq \max_i \lambda_i \tag{1.21}$$

to prevent instability (divergence). Notice that the goal here is to prevent divergence rather than preventing oscillations, this is why we have the 2 on the numerator. We can bound the rate of convergence in another direction,

$$\alpha\lambda_i < \frac{2\lambda_i}{\lambda_{\max}} \tag{1.22}$$

We call $\lambda_{\max}/\lambda_{\min}$ as ***condition number*** of $\mathbf{A}$. The larger the condition number, the slower the convergence of gradient, vice versa. Notice that $\lambda_{\max}/\lambda_{\min}$ is large when the difference between the largest smallest eigenvalue is large, i.e. the curvatures have a large difference and this is trouble-some for gradient decent updates.

Normalization In the case that we have ravine-like cost surfaces, gradient updates could have a lot of bouncing around. Even more annoyingly, gradient descent updates will calculate a larger update along the direction with steeper gradient, which in the case of ravine will almost certainly cause the gradient updates to overshoot. To overcome this, we introduce normalization, i.e. centring inputs to zero mean and unit variance. This is especially necessary when we have mixed units in inputs. The normalization we apply is

$$\tilde{x}_j = \frac{x_j - \mu_j}{\sigma_j} \tag{1.23}$$

Notice that if we want to normalize all the outputs of all the layers (including hidden layers) then we can't just normalize the output to have a mean of zero. This is due to the fact that our activations are (usually) non-zero centred, for example logistic non-linearity.

- The first trick that we can use is to replace logistic non-linearity with tanh which is also a sigmoid function, but ranges from $-1 to 1$ and thus have a zero centre.
- A recent method called ***batch normalization*** explicitly centres each hidden activation. In practice, it often speeds up training by 1.5 to 2 times!

1.5 Alternative Gradient Descent

1.5.1 Momentum Gradient Descent

To deal with ill-conditioned curvatures, we want to introduce a notion of "history" through "acceleration accumulated along the way" in our gradient descent updates. The updates are

$$\mathbf{p} \leftarrow \mu \mathbf{p} - \alpha \partial_{\boldsymbol{\theta}} \mathcal{J} \tag{1.24}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{p} \tag{1.25}$$

where α is the usually learning rate, and μ is called a damping parameter and should be some value less than 1 (typically 0.99 or 0.9). Intuitively, this is the "history decay rate" and if we have $\mu = 1$ then since we have no decay of energy through time, by conservation of energy the updates will never settle down. (which is bad, we don't want this)

- In the high curvature directions, the gradients cancel each other out, so momentum dampens the oscillations,
- In the low curvature directions, the gradient point in the same direction, allowing the parameters to "pick up speed".

- ***#! Important:*** If you increase μ, you should lower α to compensate. Here is a demonstration for the case where gradient is constant. In such case,

$$\begin{aligned} \mathbf{p} &= -\alpha \partial_{\boldsymbol{\theta}} \mathcal{J} && (1.26) \\ \implies \mathbf{p}_1 &= \mu \mathbf{p}_0 - \alpha \partial_{\boldsymbol{\theta}} \mathcal{J} = -\alpha\mu \partial_{\boldsymbol{\theta}} \mathcal{J} - \alpha \partial_{\boldsymbol{\theta}} \mathcal{J} && (1.27) \\ \implies \mathbf{p}_2 &= -\alpha\mu^2 \partial_{\boldsymbol{\theta}} \mathcal{J} - \alpha\mu \partial_{\boldsymbol{\theta}} \mathcal{J} - \alpha \partial_{\boldsymbol{\theta}} \mathcal{J} && (1.28) \\ \implies \mathbf{p}_k &= -\alpha \left(\sum_{i=1}^{k} \mu^i \right) \partial_{\boldsymbol{\theta}} \mathcal{J} \overset{k\to\infty}{\rightsquigarrow} \frac{-\alpha}{1-\mu} \partial_{\boldsymbol{\theta}} \mathcal{J} && (1.29) \end{aligned}$$

  Thus, if we want to "keep" the optimal velocity of approaching optimal spot, we will need to keep a balance between μ and α. ■

1.5.2 Stochastic Gradient Descent

The sort of gradient descent that updates each step using the entire training dataset is called ***batch training***. Alternative to that we can use SGD which updates the parameters based on the gradient for one single training example, i.e.

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla \mathcal{J}^{(i)}(\boldsymbol{\theta}) \tag{1.30}$$

this allows the training to make a lot of progress before seeing all the training data. As the name suggest, this of course will introduce noise, we can show that SGD gives unbiased estimate of the batch gradient,

$$\mathbb{E}_i \left[\nabla \mathcal{J}^{(i)}(\boldsymbol{\theta}) \right] = \frac{1}{N} \sum_{i=1}^{N} \nabla \mathcal{J}^{(i)}(\boldsymbol{\theta}) = \nabla \mathcal{J}(\boldsymbol{\theta}) \tag{1.31}$$

1.5.3 Minibatch SGD

The problem with SGD with one single training example at a time is that we can't vectorize operations and thus we can't exploit efficient vectorized operations. As a compromise, we can use mini-batch at each update, typically of size $|\mathcal{M}| = S = 100$.

- We call one pass through the *entire dataset* as a ***epoch***.
- Larger mini-batches will have smaller variance,

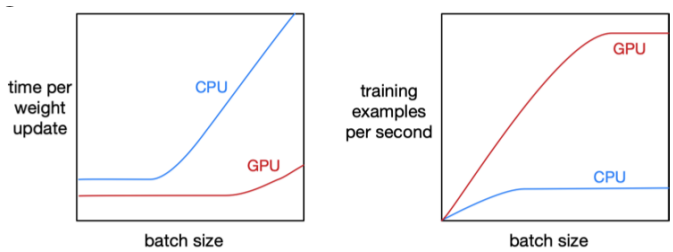
$$\mathrm{VAR}\left[\frac{1}{S} \sum_{i=1}^{S} \frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j} \right] = \frac{1}{S^2} \mathrm{VAR}\left[\sum_{i=1}^{S} \frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j} \right] = \frac{1}{S} \mathrm{VAR}\left[\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j} \right] \tag{1.32}$$

- If we choose to have larger batches, then the algorithm will converge using fewer weight updates since increasing batch size decreases the stochasticity. Otherwise, if we choose to have small batches, we will be able to perform more weight updates per second because update operations are now cheaper.

- ***#! Important:*** If actual time is proportional to FLOPs[1.3] that a processor can handle, then $S = 1$ is optimal.
  - Under this assumption, 100 updates with $S = 1$ and 1 update with $S = 100$ will require the same amount of operations and thus will require a same amount of time.
  - Clearly, holding everything else equal, we would prefer to have gradients for fresh values of parameter by using 100 updates rather than 1 update.

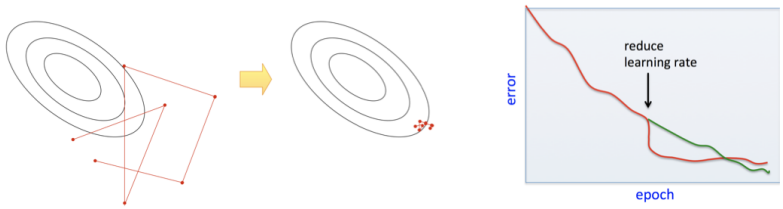
Of course, the $S = 100$ update would be less noisy, but 100 steps, albeit stochasticity, would almost certainly go further than one step.

- The batch size of choice strong depends on the hardware that the algorithm was running on[1]



1.5.4 SGD Learning Rate

- In stochastic training, the learning rate also influences the fluctuations due to the stochasticity of the gradients.
- In practice, we use a large learning rate early in training to that we can converge faster and then gradually decay the learning rate to reduce the fluctuations at the end of training.
- Notice that by reducing learning rate, we will reduce fluctuations, which can appear to make the loss drop suddenly. However, this can come at the expense of long run performance. [1]



[1.3] Floating Point Operations per second

1.6 Debugging Optimization

Here is a table that summarizes different problems that we might encounter during training time, and how to can tackle them [1]

Problem	Diagnostics	Workarounds
incorrect gradients	finite differences	fix them, or use autodiff
local optima	(hard)	random restarts
symmetries	visualize **W**	initialize **W** randomly
slow progress	slow, linear training curve	increase α; momentum
instability	cost increases	decrease α
oscillations	fluctuations in training curve	decrease α; momentum
fluctuations	fluctuations in training curve	decay α; iterate averaging
dead/saturated units	activation histograms	initial scale of **W**; ReLU
ill-conditioning	(hard)	normalization; momentum; Adam; second-order opt.

2 Convolutional Neural Net and Image Classification

2.1 Motivations for Convolution Layer

- Images are, usually, large and hence using a lot of fully connected layers would result in an insane amount of parameter to learn and makes calculation intractable.
- In images, there are usually *local* patterns or relationships. For example in the task of semantic segmentation, usually the pixel at bottom left has very little to do with the pixel at top right and using a fully connected layer where every pixel contributes to every output would be wasteful. Thus, we want to have an operation that focus on ***local*** patterns of the input image.
- Also, the same sorts of features that are useful in analyzing one part of the image will probably be useful for analyzing other parts of the image as well which motivates us in using a "filter" to "slide" across the input.

2.2 The Convolution Operator

2.2.1 1-D Signal Processing

Consider two arrays, a and b. The result of the convolution will be a new array, where

$$(a * b)_t = \sum_{\tau} a_\tau b_{t-\tau} \tag{2.1}$$

In order to make this indexing notation to work, we have to have the indices start at zero rather at one.

Here the summation over τ is a lazy notation for saying summing over all combinations that makes sense. [2.1]

[2.1] We assume infinite zero padding here, more on this later.

2.2.2 2-D Convolution

Consider two two dimensional arrays, A and B. The result of the convolution will be such that the slot at $(A * B)_{ij}$ is calculated as

$$(A * B)_{ij} = \sum_s \sum_t A_{st} B_{i-s,j-t} \tag{2.2}$$

Usually, we call the matrix/tensor that we convolve the original input with a "kernel" or "filter".

2.2.3 Properties of Convolution

- Convolution is ***Commutative***, i.e.,

$$a * b = b * a \tag{2.3}$$

- Convolution is ***Linear***, i.e.,

$$a * (\lambda_1 b + \lambda_2 c) = \lambda_1 a * b + \lambda_2 a * c \tag{2.4}$$

2.3 Canonical Kernels

Some of these canonical kernels are not working as expected...

2.3.1 Blurring Kernel

$$\begin{array}{|c|c|c|}\hline 0 & 1 & 0 \\ \hline 1 & 4 & 1 \\ \hline 0 & 1 & 0 \\ \hline \end{array} \tag{2.5}$$

2.3.2 Sharpening Kernel

$$\begin{array}{|c|c|c|}\hline 0 & -1 & 0 \\ \hline -1 & 5 & -1 \\ \hline 0 & -1 & 0 \\ \hline \end{array} \tag{2.6}$$

2.3.3 Edge Detector Kernel

$$\begin{array}{|c|c|c|}\hline 0 & -1 & 0 \\ \hline -1 & 4 & -1 \\ \hline 0 & -1 & 0 \\ \hline \end{array} \tag{2.7}$$

2.3.4 Vertical Edge Detector Kernel

$$\begin{array}{|c|c|c|}\hline 1 & 0 & -1 \\ \hline 2 & 0 & -2 \\ \hline 1 & 0 & -1 \\ \hline \end{array} \tag{2.8}$$

2.4 Convolutional Networks

In a Convolutional Neural Net, of course there will be convolution layers, however there is another sort of layers that are common, called the pooling layer. Intuitively, pooling layers shrink the dimensions by taking a "local pool".

2.4.1 Pooling Layer

Most commonly, we use the max-pooling operation in the pooling layer, which computes the maximum of the units in a pooling group

$$y_i = \overset{\max}{j \text{in local pooling group}} z_j \tag{2.9}$$

where z represents the input and y is the output. Typically, we use a 2×2 max pooling unit, which outputs

$$\begin{array}{|c|c|}\hline \alpha & \beta \\ \hline \gamma & \theta \\ \hline \end{array} \quad \longrightarrow \quad \boxed{\max\{\alpha, \beta, \gamma, \theta\}} \tag{2.10}$$

Thanks to pooling layers, deeper layers' filters will cover a larger region of the input than equal-sized filters in the lower layers. We say that deeper pooling layers have larger receptive fields in terms of the original image.

2.4.2 Non-linearity in Convolutional Layers

After convolution operation, it is common to add a non-linear activation function to introduce non-linearity. We are doing this because convolution is a linear operation and stacking convolution layers together without non-linearity is no more powerful than a single linear layer (possibly a fully connected layer). For example, the order of the layers could be

$$\text{Image} \rightarrow \underbrace{\text{Convolution} \rightarrow \text{ReLU activation}}_{\text{Convolution Layer}} \rightarrow \text{Max Pooling} \rightarrow \text{Convolution} \rightarrow \cdots \tag{2.11}$$

2.4.3 Equivariance and Invariance

TODO: I have no idea...

We want our network's responses to be robust to translations of the input, which could mean the following two things

- Convolution layers are equivariant[2.2]: if you translate the inputs, the outputs are translate by the same amount.
- Network's predictions are invariant: if you translate the inputs, the prediction should not change. Pooling layers provide invariance to small translations.

2.4.4 Channels in Convolution Layers

Each layer is consist of several feature maps, or ***channels***, each of which is an array (and of the same size). In the case where input is an image, then usually we would have 3 channels for coloured input (RGB channels) and 1 channel if it is in greyscale. Each unit is connected to each unit within its receptive field in the previous layer. This includes *all* of the previous layer's feature maps.

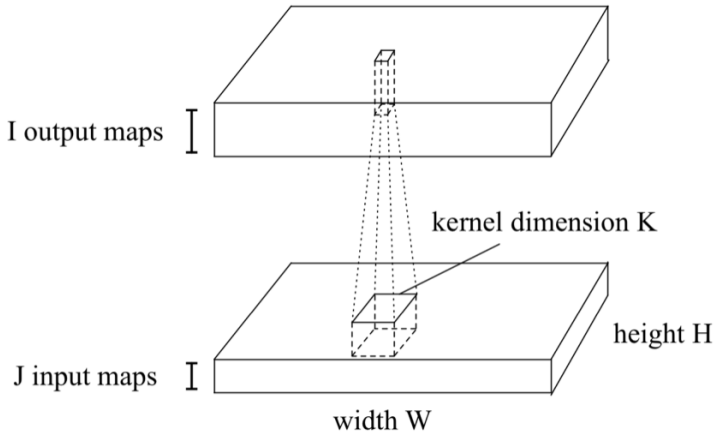
2.5 Size of Convolutional Neural Nets

There are several "measures of sizes" that are interested in, namely

- ***Number of units:*** measures the activations needed to be stored in memory during training for back propagation.
- ***Number of weights:*** since weights need to be stored (and updated at each iteration) in memory.
- ***Number of connections:*** measures the computation costs; approximately 3 add-multiply operations per connection. (1 for the forward pass, and 2 for backward pass.)

Below is the number of parameters in a fully connected multilayer perceptron layer along side with those in a convolutional layer. [2]

[2.2] Equivariant means roughly "unchanged in terms of distortion"



	fully connected layer	convolution layer
# output units	WHI	WHI
# weights	W^2H^2IJ	K^2IJ
# connections	W^2H^2IJ	WHK^2IJ

2.5.1 Rule of thumb on size

- Most of the units and connections are in the convolution layers, and
- Most of the weights are in the fully connected layers.

2.6 Supervised Pre-training and Transfer Learning

- In practice, we don't usually train an image classifier from scratch since it is unlikely hat we will have millions of cleanly labeled images for our specific datasets.
- If we want to do a computer vision task, it is common to just "fine-tune" a pre-trained convolutional neural net on ImageNet or OpenImage.
- We will fix most of the weights in the pre-trained network. Only the weights in the last layer will be randomly initialized and learnt on the current dataset/task.

Fine-tune, how?

- This depends on how many training examples do we have in the new dataset. For example, if we have fewer ew examples, then we can fix more weights from the pre-trained networks and only train a small subset of all weights. Of course, vice versa.

- This also depends on how similar is the new dataset to the dataset that our pre-trained model was trained on. That is, we need 'more' fine-tuning if they have dissimilar datasets.
- ***#! Important:*** Learning rate for the fine-tuning stage is often much lower than the learning rate used for training from scratch. why?

References

[1] Jimmy Ba. Csc413/2516 lecture 4: Optimization. 2020 Winter Term.

[2] Jimmy Ba. Csc421/2516 lecture 5: Convolutional neural network & image classification. 2020 Winter Term.