Vector Derivatives

Let $\mathbf{x}, \theta \in \mathbb{R}^n$

$\boldsymbol{\theta^T}\mathbf{x}$: $\nabla_{\mathbf{x}}\theta^T\mathbf{x} = \theta \mid \nabla_\theta \theta^T\mathbf{x} = \mathbf{x}$

$\mathbf{x^T A x}$: $\nabla_{\mathbf{x}}\mathbf{x}^T\mathbf{A}\mathbf{x} = (\mathbf{A}+\mathbf{A}^T)\mathbf{x} \mid \nabla_{\mathbf{A}}\mathbf{x}^T\mathbf{A}\mathbf{x} = \mathbf{x}\mathbf{x}^T \mid$ **A** symmetric, then $\nabla_{\mathbf{x}}\mathbf{x}^T\mathbf{A}\mathbf{x} = 2\mathbf{A}\mathbf{x}$.

Matrix Derivatives

Let $\mathbf{z}^T \in \mathbb{R}^m, \mathbf{x} \in \mathbb{R}^n, \mathbf{A} \in \mathbb{R}^{m\times n}$.

$\mathbb{R}^{m\times n} \ni \nabla_{\mathbf{A}}\mathbf{z}^T\mathbf{A}\mathbf{x} = \left[\frac{\partial \mathbf{x}^T\mathbf{A}\mathbf{x}}{\partial a_{ij}}\right]$ for $i = 1, \cdots, m$ and $j = 1, \cdots, n$

General Formula:

$y = \sum_{i=0}^n a_i x^i$. Higher degrees always fit better, but this may lead to overfitting.

Evaluating Generalization Error

Training data is what's used to learn θ. **Testing data** is data that the model has never seen before. We typically want to do better on ***testing data***. **Overfitting** is when the model has *low* training error, but *high* testing error. More data helps mitigate overfitting; *more data* $\implies$ *more complex models*. **Underfitting** is when a model has *high* training error and *high* testing error. **Hyperparameters** are parameters chosen before we start training. **Validation** data is data used to optimize the *hyperparameters*. All of these datasets are disjoint.

k-fold Cross Validation

If our **training data** has N examples, pick any $k \in \mathbb{Z}^{>0}$. Define each fold to have N/k examples. Then, $k-1$ folds are used to train the model and learn θ. The k^{th} fold is **validation data** used to evaluate the model. We can repeatedly train the model by changing which folds are for **training** and **validation**. Then $k-1$ *folds* $\to \theta, k^{th}$ *fold* $\to \mathcal{L}$.

Data Driven Approach

We *train* on data to get model parameters θ. In deep NN's, this results in learning *features* that are optimized for image classification. Here, *features*[1] $:= \hat{\mathbf{x}} = \begin{bmatrix} x^n & \cdots & 1\end{bmatrix}^T$ are inferred by the training data. We then *test* our model to classify new images.

K-Nearest Neighbors (KNN)

Find the k-closest points to $\mathbf{x}^{new}$ in the training set, according to an appropriate metric (e.g. L_2). Then the majority vote is the class $\mathbf{x}^{new}$ is assigned to. **Formally:** Define the distance metric $d(\mathbf{x}^{new}, \mathbf{x}^{(i)})$. Choose $k \in \mathbb{Z}^{>0}$. Take $d(\mathbf{x}^{new}, \mathbf{x}^{(i)})$ for $i = 1, \ldots, m$ and find the k nearest neighbors $\{c_1, \cdots c_k\}$. Take the plurality vote (randomizing ties) to classify $\mathbf{x}^{new}$. Here, the hyperparameters are $\{k, d(\cdot)\}$

Issues for Image Classification

(i) *pixel differences* $\neq$ *semantic differences*. If we transform the image, per pixel, the image is different, but the image is the same. **(ii)** The curse of dimensionality: As $n \to \infty$, the notion of "distance" becomes harder to define and volume increases exponentially.

Softmax Classifier

We want to "score" an image against each class and pick the class with the highest score.

Example: Let $\mathbf{W} := \begin{bmatrix}\mathbf{w}_1^T & \cdots & \mathbf{w}_c^T\end{bmatrix}^T \in \mathbb{R}^{10\times 3072}, c = 10, \mathbf{x} \in \mathbb{R}^{3072}, \mathbf{y}, \mathbf{b} \in \mathbb{R}^{10}$. Then $\mathbf{y}$ is the vector containing the score for class i. The chosen class is i s.t. $y^{(i)} := \max\{\mathbf{y}\}$, where $\mathbf{y}$ is

$$\mathbf{y} := \begin{bmatrix} — \mathbf{w}_1^T — \\ \vdots \\ — \mathbf{w}_{10}^T — \end{bmatrix} \begin{bmatrix} \\ \mathbf{x} \\ \\ \end{bmatrix} + \begin{bmatrix} \\ \mathbf{b} \\ \\ \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1^T\mathbf{x} + b_1 \\ \vdots \\ \mathbf{w}_{10}^T\mathbf{x} + b_1 \end{bmatrix}$$

What a Linear Classifier Does

Let $\mathbf{x} \in \mathbb{R}^2, y_1 = \mathbf{w}_1^T\mathbf{x}$. Then $\underline{y_1} = \|\mathbf{w}_1\|\|\mathbf{x}\|\cos(\theta) = \underline{\|\mathbf{x}\|\cos(\theta)}$ when $\|\mathbf{w}_1\| = 1$.

INSERT THE PICTURE

Linear classifiers fail for things like **xor**.

Chain Rule for Probability

$p(a,b) = p(a)p(b \mid a) = p(b)p(a \mid b)$

$p(a,b,c) = p(c)p(a \mid c)p(b \mid a,c) = p(a)p(b \mid a)p(c \mid a,b) = p(a,c)p(b \mid a,c)$

$p(b,c \mid d,e) = \frac{p(b,c,d,e)}{p(d)p(e|d)}$

$p(d \mid e)p(b,c \mid d,e) = \frac{p(a,b,c,d,e)}{p(a|b,c,d,e)}$

Softmax

$\text{softmax}_i(\mathbf{x}) := \frac{e^{\mathbf{w}_i^T\mathbf{x}+b_i}}{\sum_{j=1}^c e^{\mathbf{w}_j^T+\mathbf{x}+b_j}}$. If we define $a_i(\mathbf{x}) := \mathbf{w}_i^T\mathbf{x} + b_i$, we get $\text{softmax}_i(\mathbf{x}) = \frac{e^{a_i(\mathbf{x})}}{\sum_{j=1}^c e^{a_i(\mathbf{x})}}$.

$a_i(\mathbf{x})$ is the score of the image $\mathbf{x}$ being in the i^{th} class and $\text{softmax}_i(\mathbf{x})$ is the probability of $\mathbf{x}$ being in the i^{th} class.

Notation: Define $\widetilde{\mathbf{w}}_i^T := \begin{bmatrix}\mathbf{w}_i^T & b_i\end{bmatrix}, \widetilde{\mathbf{x}} := \begin{bmatrix}\mathbf{x} \\ 1\end{bmatrix}$. Then $a_i(\widetilde{\mathbf{x}}) = \widetilde{w}_i^T\widetilde{\mathbf{x}}$, $\text{softmax}_i(\widetilde{\mathbf{x}}) = \frac{e^{a_i(\widetilde{\mathbf{x}})}}{\sum_{j=1}^c e^{a_i(\widetilde{\mathbf{x}})}}$, so the probability that $\mathbf{x}^{(j)}$ belongs to class i is given by: $\Pr(y^{(j)} = i \mid \mathbf{x}^{(j)}, \theta) = \text{softmax}_i(\mathbf{x}^{(j)})$

Cross Entropy Loss

$$\mathcal{L} := \arg\min_\theta \frac{1}{m}\sum_{i=1}^m \left(\log\left[\sum_{j=1}^c e^{a_j(\mathbf{x}^{(i)})}\right] - a_{y^{(i)}}x^{(i)}\right)$$

For binary classification problems: $\mathcal{L} := -\sum_{i=1}^m (y_i \log[\sigma(z)] + (1-y_i)\log[1-\sigma(z)])$. Then the gradient is $\nabla_{\sigma(z)}\mathcal{L} = \frac{y_i}{\sigma(z)} - \frac{1-y_i}{1-\sigma(z)}$ The parameters for $\mathcal{L}$ are $\theta = \{\mathbf{W}, \mathbf{b}\}$ where $\mathbf{W} \in \mathbb{R}^{c\times n}, \mathbf{b} \in \mathbb{R}^c$.

Gradient Descent

Recall that for $\varepsilon > 0$ sufficiently small, $f(x+e) \approx f(x) + \varepsilon f'(x)$. Let $\mathcal{L} := f, x = \theta$. Then, for n-dimensional loss landscapes, we have n search dimensions to make $\mathcal{L}(\theta) \to 0$. As $n \to \infty$, the local minima $\to$ global minimum. So $\mathcal{L}(\theta + \Delta\theta) \approx \mathcal{L}(\theta) + \Delta\theta^T\nabla_\theta\mathcal{L}(\theta)$. To minimize $\mathcal{L}$, we need to find $\min\limits_{\mathbf{u}, \|\mathbf{u}\|=1} \mathbf{u}^T\nabla_\theta\mathcal{L}(\theta) = \min\limits_{\mathbf{u}, \|\mathbf{u}\|=1} \|\mathbf{u}\|\|\nabla_\theta\mathcal{L}(\theta)\|\cos(\theta) = \min\limits_{\mathbf{u}} \|\nabla_\theta\mathcal{L}(\theta)\|\cos(\theta)$ which is minimized when $\cos(\theta) = -1$, so $\mathbf{u} := -\nabla_\theta\mathcal{L}(\theta)$. Then we repeatedly calculate $\theta \leftarrow \theta - \varepsilon\nabla_\theta\mathcal{L}(\theta)$.

Why Not Numerical? There are too many parameters. Takes too much time.

Intuition: The gradient w.r.t. the parameters is a function of the training data. We can think of each point as a noisy estimate of the gradient at that point.

Batch v. Minibatch Given m examples:

Batch: uses all m examples in the training data to calculate the gradient. **Minibatch:** approximates the gradient by using k examples for computation, where $1 < k < m$. **Stochastic:** approximates the gradient over one example. We typically use minibatching for neural networks. Minibatch *may* be referred to SGD.

Nomenclature: The **input layer** is the first layer of the NN, typically $\mathbf{x}$. The **output layer** is the last layer of the NN, typically $\mathbf{z}$. The **hidden layers** are the intermediate layers of the NN, typically $\mathbf{h}_i$. For a NN with N layers, we ***do not*** count the input layer as part of N. Note that $\mathbf{z}$ are the scores that go a softmax classifier, sometimes called "logits".

Example

Let $\underline{\mathbf{h}_1} = \mathbf{W}_1\mathbf{x} + \underline{\mathbf{b}_1,}$ $\mathbf{h}_i = \mathbf{W}_i\mathbf{h}_{i-1} + \underline{\mathbf{b}_1}$ for $i = 2, \cdots, N-1$, $\mathbf{z} = \mathbf{W}_N\mathbf{h}_{N-1} + \mathbf{b}_N$. Then $\mathbf{z} = \widetilde{\mathbf{W}}\mathbf{x} + \widetilde{\mathbf{b}}$ where $\widetilde{\mathbf{W}} = \mathbf{W}_N \cdots \mathbf{W}_1$ and $\widetilde{\mathbf{b}} = \mathbf{b}_N + \sum_{i=2}^{N-1}\mathbf{W}_i\mathbf{b}_{i-1}$.

Linear *f*: $f(x) = ax + b$ can be useful if $\dim(\mathbf{h}) \ll \dim(\mathbf{x})$. This corresponds to finding a low-rank representation of the inputs (e.g. autoencoder).

f is the **activation function** and is typically ***not*** applied to $\mathbf{z}$.

Hidden Layers as Learned Features: Nonlinear $f(\mathbf{h}_i)$ finds features of the data. If $\text{softmax}(\mathbf{z})$ is good, then $\mathbf{h}_i$ is linearly separable.

Learnable Parameters: $\left(|\mathbf{x}| \cdot |\mathbf{h}_1| + \left(\sum_{i=1}^{N-2}|\mathbf{h}_i| \cdot |\mathbf{h}_{i+1}|\right) + \mathbf{h}_{N-1} \cdot |\mathbf{z}|\right) + \left(|\mathbf{z}| + \sum_{i=1}^{N-1}|\mathbf{h}_i|\right)$.

Activation Functions

__**Sigmoid:**__ $\sigma(x) = \frac{1}{1+e^{-x}}$. $\nabla_x\sigma = \sigma(x)(1-\sigma(x))$.

Vanishing Gradient Problem: We want $\nabla_{\mathbf{w}}\mathcal{L}$ to be large since $\mathbf{w} \leftarrow \mathbf{w} - \varepsilon\nabla_{\mathbf{w}}\mathcal{L}$. Then we have $f(\mathbf{w}_1^T\mathbf{x} + \mathbf{b}) =: \sigma(\mathbf{w})$, so $\frac{\partial\mathcal{L}}{\partial\mathbf{w}} = \frac{\partial\sigma}{\partial\mathbf{w}} \cdot \frac{\partial\mathcal{L}}{\partial\sigma}$. For extreme inputs, there is no learning since $\frac{\partial\mathcal{L}}{\partial\mathbf{w}} \approx 0$, so $\mathbf{w} \leftarrow \mathbf{w} + 0$ (saturation). σ can have zig-zagging gradients since σ is always nonnegative.

Output Activations: $\hat{y}^{(i)} = \text{softmax}_i(\mathbf{z})$ is the generalization of σ to multiple classes.

__**ReLu:**__ $\text{relu}(x) = \max\{0, x\}$. $\nabla_x\text{relu} = \begin{cases} 1 & x > 0 \\ 0 & else\end{cases}$. ReLu is widely used in practice. It can still have zig-zagging gradients but there is no saturation.

__**tanh:**__ $\tanh(x) = 2\sigma(2x) - 1$. $\nabla_x\tanh = 1 - \tanh^2(x)$ It is zero-centered so there is no vanishing gradient problem, but there's still saturation for extreme inputs.

[1] $\hat{\mathbf{x}}$ is the vectorzed version of $1, \cdots x^n$ in $\mathbf{y} := \sum_{i=0}^n a_i x^i$

Deep Learning Architecture: *Input* $\to$ *Hidden* $\to$ *Output* $\to$ softmax $\to \mathcal{L}$. **Loss function:** $\text{softmax}_i(\mathbf{z})$. **Learning role:** $\theta \leftarrow \theta - \varepsilon \nabla_\theta \mathcal{L}$.

Nomenclature

Forward pass is $\mathbf{x} \to \cdots \to \mathbf{z} \to \mathcal{L}$. **Backpropagation** is $\frac{\partial \mathcal{L}}{\partial \mathbf{z}} \to \cdots \to \frac{\partial \mathcal{L}}{\partial \theta_1}$. Backpropagation operationalizes the gradient. It is computationally efficient since we cache values during forward pass to use during the backward pass. In general, we have

$$\begin{tikzcd}
\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial f}{\partial x}\frac{\partial \mathcal{L}}{\partial f} \arrow[dr, "x"] & & \\
& f \arrow[r, "f(x,y)"] & \frac{\partial \mathcal{L}}{\partial z} \\
\frac{\partial \mathcal{L}}{\partial y} = \frac{\partial f}{\partial y}\frac{\partial \mathcal{L}}{\partial f} \arrow[ur, "y"'] & &
\end{tikzcd}$$

where $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}$ are local gradients and $\frac{\partial \mathcal{L}}{\partial f}$ is the upstream gradient.

Gates

Addition: Distribute the upstream gradient $\frac{\partial \mathcal{L}}{\partial f}$.

$$\begin{tikzcd}
\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial f} \arrow[dr, "x"] & & \\
& + \arrow[r, "f := x+y"] & \frac{\partial \mathcal{L}}{\partial f} \\
\frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial f} \arrow[ur, "y"'] & &
\end{tikzcd}$$

Multiplication (Scalar): Switch x, y and multiply by the upstream gradient $\frac{\partial \mathcal{L}}{\partial f}$.

$$\begin{tikzcd}
\frac{\partial \mathcal{L}}{\partial x} = y \cdot \frac{\partial \mathcal{L}}{\partial f} \arrow[dr, "x"] & & \\
& \times \arrow[r, "f := x \cdot y"] & \frac{\partial \mathcal{L}}{\partial f} \\
\frac{\partial \mathcal{L}}{\partial y} = x \cdot \frac{\partial \mathcal{L}}{\partial f} \arrow[ur, "y"'] & &
\end{tikzcd}$$

Multiplication (n-Dimensional Vector/Tensor): Fancy tensor derivative shortcut.

$$\begin{tikzcd}
\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Y}}\mathbf{X}^T \arrow[dr, "\mathbf{W}"] & & \\
& \times \arrow[r, "\mathbf{Y} := \mathbf{W}\mathbf{X}"] & \frac{\partial \mathcal{L}}{\partial f} \\
\frac{\partial \mathcal{L}}{\partial \mathbf{X}} = \mathbf{W}^T \frac{\partial \mathcal{L}}{\partial \mathbf{Y}} \arrow[ur, "\mathbf{X}"'] & &
\end{tikzcd}$$

Max: Route the upstream gradient $\frac{\partial \mathcal{L}}{\partial f}$.

$$\begin{tikzcd}
\frac{\partial \mathcal{L}}{\partial x} = \mathbb{I}_{\{y<x\}} \odot \frac{\partial \mathcal{L}}{\partial f} \arrow[dr, "x"] & & \\
& \max \arrow[r, "f := \max\{x,y\}"] & \frac{\partial \mathcal{L}}{\partial f} \\
\frac{\partial \mathcal{L}}{\partial y} = \mathbb{I}_{\{x<y\}} \odot \frac{\partial \mathcal{L}}{\partial f} \arrow[ur, "y"'] & &
\end{tikzcd}$$

ReLu: Hadamard product the upstream gradient $\frac{\partial \mathcal{L}}{\partial f}$.

$$\begin{tikzcd}
\frac{\partial \mathcal{L}}{\partial x} = \mathbb{I}_{\{\mathbf{x}>0\}} \odot \frac{\partial \mathcal{L}}{\partial f} \arrow[r, "\mathbf{x}"] & \text{relu} \arrow[r, "f := \text{relu}(\mathbf{x})"] & \frac{\partial \mathcal{L}}{\partial f}
\end{tikzcd}$$

$\boldsymbol{f^{-1}}$**:** Derivative of $^1/_x$ multiplied by the upstream gradient $\frac{\partial \mathcal{L}}{\partial f}$.

$$\begin{tikzcd}
\frac{\partial \mathcal{L}}{\partial x} = -\frac{1}{x^2} \cdot \frac{\partial \mathcal{L}}{\partial f} \arrow[r, "x"] & f^{-1} \arrow[r, "f := f^{-1}(x)"] & \frac{\partial \mathcal{L}}{\partial f}
\end{tikzcd}$$

$\boldsymbol{e^x}$**:** Derivative of e^x multiplied by the upstream gradient $\frac{\partial \mathcal{L}}{\partial f}$.

$$\begin{tikzcd}
\frac{\partial \mathcal{L}}{\partial x} = e^x \cdot \frac{\partial \mathcal{L}}{\partial f} \arrow[r, "x"] & e^x \arrow[r, "f := e^x"] & \frac{\partial \mathcal{L}}{\partial f}
\end{tikzcd}$$

$\boldsymbol{x^n}$**:** Derivative of x^n multiplied by the upstream gradient $\frac{\partial \mathcal{L}}{\partial f}$.

$$\begin{tikzcd}
\frac{\partial \mathcal{L}}{\partial x} = nx^{n-1} \cdot \frac{\partial \mathcal{L}}{\partial f} \arrow[r, "x"] & x^n \arrow[r, "f := x^n"] & \frac{\partial \mathcal{L}}{\partial f}
\end{tikzcd}$$

$\boldsymbol{\sqrt{x}}$**:** Derivative of $\sqrt{x}$ multiplied by the upstream gradient $\frac{\partial \mathcal{L}}{\partial f}$.

$$\begin{tikzcd}
\frac{\partial \mathcal{L}}{\partial x} = \frac{1}{2\sqrt{x}} \cdot \frac{\partial \mathcal{L}}{\partial f} \arrow[r, "x"] & \sqrt{x} \arrow[r, "f := \sqrt{x}"] & \frac{\partial \mathcal{L}}{\partial f}
\end{tikzcd}$$

Law of Total Derivatives

$$\begin{tikzcd}
& \frac{\partial \mathcal{L}}{\partial q_1} \arrow[dr, "q_1"] & & \\
\frac{\partial \mathcal{L}}{\partial x} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}}{\partial q_i} \arrow[ur, "x"] \arrow[dr, "x"'] & \vdots & \frac{\partial \mathcal{L}}{\partial f} \arrow[r, "f"] & \cdots \\
& \frac{\partial \mathcal{L}}{\partial q_n} \arrow[ur, "q_n"'] & &
\end{tikzcd}$$

Multivariate Chain Rule: $\nabla_\mathbf{x}\mathbf{z} = \nabla_\mathbf{x}\mathbf{y} \cdot \nabla_\mathbf{y}\mathbf{z} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{y}} = \frac{\partial \mathbf{z}}{\partial \mathbf{x}}$

Regularization

Initialization: Weight initialization can heavily impact the performance of a NN. **Small** random weight initialiations cause all activations to go to 0 since $\frac{\partial \mathcal{L}}{\partial \theta_i} \to 0$. **Large** random weight initializations cause all activations to go to ∞ since $\frac{\partial \mathcal{L}}{\partial \theta_i} \to \infty$.

Xavier Initialization

Try to keep the variences between layers equal: $\text{var}(h_i) \approx \text{var}(h_j) \implies \text{var}(\nabla_{h_i} L) \approx \text{var}(\nabla_{h_j} \mathcal{L})$.

Derivation: Suppose all units are linear; i.e. $h_i := \sum_{j=1}^{n_{in}} w_{ij} h_{i-1,j}$ and suppose $w_{ij}, h_{i-1,j}$ are independent. Then $\underline{\text{var}(wh)} = \mathbb{E}^2(w)\,\text{var}(h) + \mathbb{E}^2(h)\,\text{var}(w) + \text{var}(w)\,\text{var}(h) = \underline{\text{var}(w)\,\text{var}(h)}$ if $\mathbb{E}(w) = \mathbb{E}(h) = 0$. Then $\underline{\text{var}(h_i)} = \text{var}\left(\sum_{j=1}^{n_{in}} w_{ij} h_{i-1,j}\right) = \underline{\text{var}(h_{i-1}) \cdot \sum_{j=1}^{n_{in}} \text{var}(w_{ij})} \implies \sum_{j=1}^{n_{in}} \text{var}(w_{ij}) = 1$ to get $\text{var}(h_i) = \text{var}(h_{i-1})$. Assuming the weights have the same statistics, $n_{in} \cdot \text{var}(w) = 1 \implies \text{var}(w) = \frac{1}{n_{in}}$. So $\text{var}(w_{ij}) = \frac{1}{n_{in}}$. Similarly for backprop, $\text{var}(w_{ij}) = \frac{1}{n_{out}}$, so $\text{var}(w_{ij}) = \frac{2}{n_{in}+n_{out}} = \frac{1}{n_{avg}}$.

Batch Normalization

We want to avoid saturation and high variance in activiations. The issue is that the changes we make at each gradient step affects the other layers. For example, $\mathbf{W}_3 \leftarrow \mathbf{W}_3 - \varepsilon \frac{\partial \mathcal{L}}{\partial \mathbf{W}_3}$ changes $\mathbf{h}_3$ based on $\mathbf{h}_2$, but $\mathbf{h}_2$ may also change due to $\mathbf{W}_2 \leftarrow \mathbf{W}_2 - \varepsilon \frac{\partial \mathcal{L}}{\partial \mathbf{W}_3}$. By induction, we see that the issue propagates. The idea of **batch normalization** is to make the output of each layer have *unit statistics*: $\mathbf{h}_i = \text{relu}(x_i), \mathbb{E}(x_i) = 0, \text{var}(x_i) = 1$.

Vanilla: *affine* $\to$ relu $\to$ *affine* $\to$ relu

Batch Normalization: *affine* $\to$ batch-norm $\to$ relu $\to$ *affine* $\to$ batch-norm $\to$ relu

Other Bullshit (HW): *affine* $\to$ relu $\to$ batch-norm $\to$ *affine* $\to$ relu $\to$ batch-norm

Batch norm meets the ***Xavier Initialization*** criteria ($\text{var}(h_i) \approx \text{var}(h_j)$), so you don't need to use Xavier but it's still good to.

Normalizing Unit Activations

Given a batch of m samples, the **normalized value** of the i^{th} activation is defined as $\hat{x}_i := \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}$ where $\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)}$ is the **mean** and $\sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2$ is the **variance** of the activations across the batch. We then ***scale*** and ***shift*** the normalized activations using learned parameters γ and β respectively, to get the **output** $y_i = \gamma_i \hat{x}_i + \beta_i$.

Training: During *training*, we normalize the output of a layer for each minibatch. We also compute a running mean and varience of the past k minibatches for $1 \le k \le N$. We learn γ and β via ***backpropagation*** in addition to θ.

Testing: During *testing*, we use the running mean and variance computed during *training* to normalize each layer. Additionally, we use the γ and β that were learned during *training*.

Regularization is any modification that improves testing/validation error but doesn't decrease training error. (e.g. Stopping early)

Parameter Norm Penalty

The **parameter norm penalty is** denoted as $\Omega(\theta)$ where Ω is a hyperparameter like $\mathcal{L}$. The cost function then becomes $\mathcal{L} := \mathcal{L}(\theta \mid \mathbf{X}, \mathbf{y}) + \alpha\Omega(\theta)$ where $\alpha \ge 0$ weights $\Omega(\theta)$.

***L*₂ Regularization**

$\Omega(\theta) = \frac{1}{2}\mathbf{w}^T\mathbf{w} = \frac{1}{2}\|\mathbf{w}\|^2$ promotes models with parameters close to 0. That is, we want the norms to be small. $\frac{\partial \Omega}{\partial \mathbf{W}} = \mathbf{W}$, so our gradient step looks like $\widetilde{\mathcal{L}}(\theta \mid \mathbf{X}, \mathbf{y}) = \mathcal{L}(\theta \mid \mathbf{X}, \mathbf{y}) + \frac{\alpha}{2}\mathbf{w}^T\mathbf{w} \to \nabla_\mathbf{w}\widetilde{\mathcal{L}} = \nabla_\mathbf{w}\widetilde{\mathcal{L}} + \alpha\mathbf{w} \implies \mathbf{w} \leftarrow \mathbf{w} - \varepsilon\nabla_\mathbf{w}\widetilde{\mathcal{L}} \iff \mathbf{w} \leftarrow (1 - \varepsilon\alpha)\mathbf{w} - \varepsilon\nabla_\mathbf{w}\mathcal{L}$. We want small weights because large weights are more sensitive to variable inputs which leads to overfitting.

Extensions: If we know $\mathbf{w}$ is close to some $\mathbf{b}$, set $\Omega(\theta) = \|\mathbf{w} - \mathbf{b}\|_2^2$. More generally, if we need two weights $\mathbf{w}^{(i)}, \mathbf{w}^{(j)}$ to be close, set $\Omega(\theta) = \|\mathbf{w}^{(i)} - \mathbf{w}^{(j)}\|_2^2$

***L*₁ Regularization**

$\Omega(\theta) = \|\mathbf{w}\|_1 = \sum_i |w_i|$. This may be used for feature selection by pruning weights that are 0.

Sparse Representation

$\Omega(\mathbf{h}_i) = \|\mathbf{h}_i^{(i)}\|$.

Dataset Augmentation
We can **augment** (e.g. crop, reflection, gaussian blur, etc.) the original image(s) to increase the size of the training data and also make the NN more robust to augmented data.
Label Smoothing: Adding noise helps to reduce error. We go from a *one hot* representation to **label smoothing**: $\mathcal{L}^{(i)} = \sum_{c=1}^{C} -y_c \log(p_c) \to \mathcal{L}^{(i)} = \sum_{c=1}^{C} -y_c(1-\alpha) + \frac{\alpha}{C-1}$ where α is a hyperparameter.

Transfer Learning
Training a NN in one context and using them in another with minimal additional training.
Ensemble Methods
Training multiple different models and averaging their results at test time.
Intuition: Independent models make independent errors.
Why this works: For one model, $\mathbb{E}(\varepsilon^2)$. Then, $\mathbb{E}(\varepsilon_i\varepsilon_j) = \mathbb{E}(\varepsilon_i)\mathbb{E}(\varepsilon_j)$ if $\varepsilon_i \perp\!\!\!\perp \varepsilon_j$. Then $\mathbb{E}\left[\left(\frac{1}{k}\sum_{i=1}^{k}\varepsilon_i\right)^2\right] = \left(\frac{1}{k}\sum_{i=1}^{k}\mathbb{E}(\varepsilon_i)\right)^2 = \frac{1}{k}\mathbb{E}(\varepsilon_i)^2$. If $\varepsilon_i \not\perp\!\!\!\perp \varepsilon_j$, then $\frac{1}{k}\mathbb{E}(\varepsilon_i)^2 + \frac{k-1}{k}\mathbb{E}(\varepsilon_i\varepsilon_j)^2$ If the models are perfectly correlated, we get $\mathbb{E}(\varepsilon_i)^2$. So we are bounded below and above.

Bagging (Bootstrap Aggregating)
Constuct k datasets by drawing N examples *with* replacement for each $i = 1, \cdots, k$. Train k models and ***ensemble***.

Dropout Set a hyperparameter p to be the probability of keeping a neuron in a layer. On any *training* iteration, draw a sample *binary mask* $\mathbf{m}$. Then $\mathbf{h} \odot \mathbf{m}$ "drops" $h_j \in \mathbf{h}$ where $m_j = 0$. At *test* time, we do $\mathbf{h}_{out} = \text{relu}\left(p \cdot \sum_{i=1}^{n} w_i h_i\right)$.
Inverted dropout scales $\mathbf{m}$ by $1/p$ during *training* so we don't do anything during *testing*.

Optimization
Recall **SGD:** $g = \nabla_\theta \mathcal{L}(\theta), \theta \leftarrow \theta - \varepsilon\nabla_\theta\mathcal{L}(\theta)$
Recall **Minibatch GD:** $g = \frac{1}{m}\sum_{i=1}^{m}\nabla_\theta\mathcal{L}(\theta)$, $\theta \leftarrow \theta - \varepsilon g$.
Momentum
We maintain a *running mean* of the gradients. Initialize $\mathbf{v} = 0, \alpha \in [0,1]$ (typically 0.99). Then
(1) Compute: $\mathbf{g}$ **(2)** Update: $\mathbf{v} \leftarrow \alpha\mathbf{v} - \varepsilon\mathbf{g}$ **(3)** Step: $\theta \leftarrow \theta + \mathbf{v}$
The general formula is $\mathbf{v}_k := -\varepsilon\sum_{i=1}^{k}\alpha^{k-1}g_i$.
Momentum tends to find *better* local optima since it can take us out of bad ones. The optima that we find ourselves in tend to be *shallow* and *flat*, which means we are more resistant to changes in θ. **Nesterov Momentum**
Intuition: If $\alpha\mathbf{v}$ is good anyways, we should compute $\mathbf{g}$ *after* $\alpha\mathbf{v}$. Initialize $\mathbf{v} = 0, \alpha \in [0,1]$ (typically 0.99). Then
(2) Update: $\mathbf{v} \leftarrow \alpha\mathbf{v} - \varepsilon\nabla_\theta\mathcal{L}(\theta + \alpha\mathbf{v})$ **(3)** Gradient step: $\theta \leftarrow \theta + \mathbf{v}$
which is equivalent to
(2) Update: $\mathbf{v}' \leftarrow \alpha\mathbf{v} - \varepsilon\nabla_{\widetilde{\theta}}\mathcal{L}(\widetilde{\theta})$ **(3)** Step: $\widetilde{\theta} \leftarrow \widetilde{\theta} + \mathbf{v}' + \alpha(\mathbf{v} - \mathbf{v})$ **(4)** Next: $v' \leftarrow v, \widetilde{\theta}' \leftarrow \widetilde{\theta}$ where $\widetilde{\theta} = \theta + \alpha\mathbf{v}$. Momentum is a **first moment** update.
Adaptive Gradients (Adagrad)
Intuition: As we get closer to the local optima, we want $\varepsilon \to 0$. Initialize $\mathbf{a} = 0, \nu \approx 0$. Then
(1) Compute: $\mathbf{g}$ **(2)** Update: $\mathbf{a} \leftarrow \mathbf{a} + \mathbf{g} \odot \mathbf{g}$ **(3)** Step: $\theta \leftarrow \theta + \frac{\varepsilon}{\sqrt{\mathbf{a}}+\nu} \odot \mathbf{g}$
The above says that the *more* we step in a certain direction, the *less* impact it should have on future steps. **The issue** is that $\mathbf{a}$ is monotonically nondecreasing.
RMSProp
Fixes the issue in ***Adagrad***. Initialize $\mathbf{a} = 0, \nu \approx 0, \beta \in [0,1]$ (typically 0.99). Then
(1) Compute: $\mathbf{g}$ **(2)** Update: $\mathbf{a} \leftarrow \beta\mathbf{a} + (1-\beta)\mathbf{g} \odot \mathbf{g}$ **(3)** Step: $\theta \leftarrow \theta + \frac{\varepsilon}{\sqrt{\mathbf{a}}+\nu} \odot \mathbf{g}$
RMSProp with Momentum
Initialize $\mathbf{a} = \mathbf{v} = 0, \nu \approx 0, \alpha, \beta \in [0,1]$ (typically 0.99). Then
(1) Compute: $\mathbf{g}$ **(2)** Accumulate: $\mathbf{a} \leftarrow \beta\mathbf{a} + (1-\beta)\mathbf{g} \odot \mathbf{g}$ **(3)** Momentum: $\mathbf{v} \leftarrow \alpha\mathbf{v} - \frac{\varepsilon}{\sqrt{\mathbf{a}}+\nu} \odot \mathbf{g}$
(4) Step: $\theta \leftarrow \theta + \mathbf{v}$
Adaptive Moments (Adam)
Let $\mathbf{v}$ be the ***first moment***, $\mathbf{a}$ be the ***second moment***. Initialize $\mathbf{a} = \mathbf{v} = 0, \nu \approx 0, \beta_1, \beta_2 \in [0,1]$ (typically 0.99). Then
(1) Compute: $\mathbf{g}$ **(2)** First Moment: $\mathbf{v} \leftarrow \beta_1\mathbf{v} + (1-\beta_1)\mathbf{g}$
(3) Second Moment: $\mathbf{a} \leftarrow \beta_2\mathbf{a} + (1-\beta_2)\mathbf{g} \odot \mathbf{g}$ **(4)** Step: $\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\mathbf{a}}+\nu} \odot \mathbf{v}$

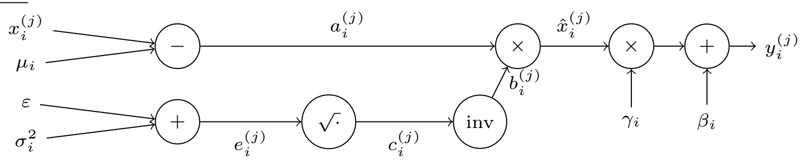
Adaptive Moments with Bias Correction (Cooler Adam)
Let $\mathbf{v}$ be the ***first moment***, $\mathbf{a}$ be the ***second moment***, and t be the iteration. Initialize $t = 0, \mathbf{a} = \mathbf{v} = 0, \nu \approx 0, \beta_1, \beta_2 \in [0,1]$ (typically 0.99). Then
(1) Compute: $\mathbf{g}$ **(2)** Time Update: $t \leftarrow t+1$ **(2)** First Moment: $\mathbf{v} \leftarrow \beta_1\mathbf{v} + (1-\beta_1)\mathbf{g}$
(3) Second Moment: $\mathbf{a} \leftarrow \beta_2\mathbf{a} + (1-\beta_2)\mathbf{g} \odot \mathbf{g}$ **(4)** Bias Correction: $\widetilde{\mathbf{v}} = \frac{1}{1-\beta_1^t}\mathbf{v}$, $\widetilde{\mathbf{a}} = \frac{1}{1-\beta_2^t}\mathbf{a}$
(4) Step: $\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\widetilde{\mathbf{a}}}+\nu} \odot \widetilde{\mathbf{v}}$
Note: All greek letters are hyperparameters $(\alpha, \beta, \beta_1, \beta_2, \varepsilon, \nu)$. Additionally, all discussed optimizers are ***first order methods***; i.e. we only use the first derivative.

Challenges in Gradient Descent
Exploding Gradients: Sometimes, the loss function can have "cliffs" where small changes drastically change $\mathcal{L}$. Because the gradient at the cliff is large, an update can result in going to a completely differen parameter space. This can be fixed using ***gradient clipping***, which upperbounds the maximum gradient norm. That is, $\|\mathbf{g}\| > clip$ so $\mathbf{g} \leftarrow \mathbf{g} \cdot {}^{clip}/_{\|\mathbf{g}\|}$.
Vanishing Gradients: Similarly, we can have vanishing gradients by repeatedly multiplying $\mathbf{W}$. By layer t, we have $\mathbf{W}^t$ multiplications. If $\mathbf{W}\mathbf{U}\Lambda\mathbf{U}^{-1}$ is its eigendecomposition, then $\mathbf{W}^t = \mathbf{U}\Lambda^t\mathbf{U}^{-1}$, so the gradient along $\mathbf{u}_i$ is grown/shrunken by a factor of λ_i^t. Architectural decisions and regularization can fix this. That is, it's a skill issue.

Convolutional Neural Networks Motivation: For images, FC NN's require many parameters. In CIFAR-10, the input size is 3072, so we need 3072 weights for each neuron. For a normal image of size 200×200, each neuron in the first layer would require 120000 parameters. This may lead to overfitting since the number of parameters $\uparrow \implies$ overfitting.
Convolution
Valid Convolution is defined as $(f \star g)(n) := \sum_{m=-\infty}^{\infty} f(m)g(n+m)$. We take our filter and start in the top left, doing point-wise element multiplication and summing up the products. The output is $\left(\frac{w-w_f+2pad}{stride}+1, \frac{h-h_f+2pad}{stride}+1\right)$ where w, h, f are the ***width***, ***height***, and ***filter*** respectively. Note that ***the depth of the filter matches the depth of the input***. For n_f filters, our output is the output of every filter applied and then stacked n_f times. Each goes through ***ReLu***.
Number of Parameters: We have $(w_f \cdot h_f \cdot d + 1) \cdot n_f$ parameters with n_f filters.
Number of Neurons We have $\frac{w-w_f+2pad}{stride} + 1 \cdot \frac{h-h_f+2pad}{stride} + 1 \cdot n_f$ neurons for n_f filters with dimensions w_f, h_f. Every time we stack units, we increase the ***receptive field***, so if we want the entire image to be seen, we add more layers.

Midterm Review
Is x^3 a good activation function? This activation function is nonlinear and differentiable everywhere, which satisfy some requirements of a good activation function. However, it is likely to cause exploding gradients as the gradient can be very large for inputs with large absolute values. Also, small inputs can lead to vanishing gradients, e.g. inputs that are close to zero
Batchnorm Train v. Test: During training, it first computes the mean and variance of the minibatch data in the unitwise. Then BatchNorm normalizes the layer's input based on the mean and variance. After the normalization, BatchNorm applies two learnable parameters for each unit: γ for scaling and β for shifting, which are learned during training and allow the network to adaptively adjust the output distribution. Meanwhile, it also keeps tracking the running averages for mean and variance through all batches. During testing, BatchNorm uses the fixed accumulated running averages from training for normalizing the testing data. The learned scaling and shifting parameters and are then applied to the normalized data.
Increase number of neurons in layers: (i) If the model is originally underfitting on the training data, adding more units in layers allows the MLP to capture more complex patterns in the data, which can improve the model performance, e.g. decrease both training and testing error. **(ii)** On the other hand, it may also cause overfitting as increasing the model's capacity is likely to make the model sensitive to the training data. As a result, the training loss may still keep decreasing while the testing error increase
Why bias correction? The estimations are biased towards 0 at the start of training because we initialize them to zero. So, the optimizer is likely to take larger steps in the first couple of updates of the model, leading to unstable training and slower convergence. Bias correction adjusts these estimations to be more accurate. After the early phase of training, the estimations tend to be accurate, so β_1, β_2 gradually approach 1, reducing the impact of bias correction.

Batchnorm



Then omitting the subscript i,

$$\frac{\partial \mathcal{L}}{\partial \beta} = \sum_{j=1}^{m} \frac{\partial \mathcal{L}}{\partial y^{(j)}}$$

$$\frac{\partial \mathcal{L}}{\partial \gamma} = \sum_{j=1}^{m} \frac{\partial \mathcal{L}}{\partial y^{(j)}} \hat{x}^{(j)}$$

$$\frac{\partial \mathcal{L}}{\partial \hat{x}^{(j)}} = \frac{\partial \mathcal{L}}{\partial y^{(j)}}$$

$$\frac{\partial \mathcal{L}}{\partial a^{(j)}} = \frac{1}{\sqrt{\sigma^2+\varepsilon}} \frac{\partial \mathcal{L}}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \mathcal{L}}{\partial \mu} = \frac{1}{\sqrt{\sigma^2+\varepsilon}} \sum_{j=1}^{m} \frac{\partial \mathcal{L}}{\partial \hat{x}^{(j)}} - \frac{\partial \mathcal{L}}{\partial \sigma^2} \frac{1}{m} \sum_{j=1}^{m} \frac{2(\mathbf{x}^{(j)}-\mu)}{m}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(j)}} = \left(x^{(j)}-\mu\right) \frac{\partial \mathcal{L}}{\partial \hat{x}^{(j)}}$$

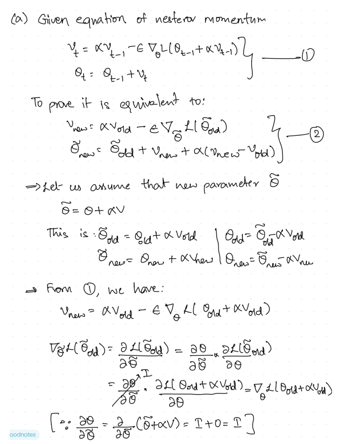
$$\frac{\partial \mathcal{L}}{\partial c^{(j)}} = -\frac{1}{\sqrt{\sigma^2+\varepsilon}} \left(x^{(j)}-\mu\right) \frac{\partial \mathcal{L}}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \mathcal{L}}{\partial e^{(j)}} = -\frac{1}{2(\sigma^2+\varepsilon)^{3/2}} \frac{\partial \mathcal{L}}{\partial \hat{x}^{(j)}}$$

$$\frac{\partial \mathcal{L}}{\partial \sigma^2} = \sum_{j=1}^{m} \frac{\partial \mathcal{L}}{\partial e^{(j)}}$$

$$\frac{\partial \mathcal{L}}{\partial \hat{x}^{(j)}} = \frac{\partial \mathcal{L}}{\partial a^{(j)}} + \frac{\partial \sigma^2}{\partial x^{(j)}} \frac{\partial \mathcal{L}}{\partial \sigma^2} + \frac{\partial \mu}{\partial x^{(j)}} \frac{\partial \mathcal{L}}{\partial \mu}$$

$\widetilde{\theta} = \theta + \alpha \mathbf{v}$. Then $\theta_{old,new} = \widetilde{\theta}_{old,new} - \alpha \mathbf{v}_{old,new}$ Then $v_{new} = \alpha \mathbf{v}_{old} - \varepsilon \nabla_\theta \mathcal{L}(\theta_{old} + \alpha \mathbf{v}_{old})$ so $\frac{\partial \mathcal{L}(\widetilde{\theta}_{old})}{\partial \widetilde{\theta}} = \frac{\partial \theta}{\partial \widetilde{\theta}} \frac{\partial \mathcal{L}(\widetilde{\theta}_{old})}{\partial \theta} = \frac{\partial \mathcal{L}(\widetilde{\theta}_{old})}{\partial \theta}$. So $\widetilde{\theta}_{old} - \alpha \mathbf{v}_{new} = \theta_{old} - \alpha \mathbf{v}_{old} + \alpha + \mathbf{v}_{new}$ becomes $\widetilde{\theta}_{old} = \theta_{old} + \mathbf{v}_{new} + \alpha(\mathbf{v}_{new} - \mathbf{v}_{old})$, so we are done.
$\widetilde{\mathcal{L}}(\theta \mid \mathbf{X}, \mathbf{y}) = \mathcal{L}(\theta \mid \mathbf{X}, \mathbf{y} + \frac{\alpha}{2}\|\theta\|_2^2$. Then $\nabla_\theta \widetilde{\mathcal{L}} = \nabla_\theta \mathcal{L} = \nabla_\theta \mathcal{L} + \alpha\theta$. Then $\theta \leftarrow \theta - \varepsilon \nabla_\theta \widetilde{\mathcal{L}} \implies (1-\varepsilon\alpha)\theta - \varepsilon(\nabla_\theta \mathcal{L}(\theta \mid \mathbf{X}, \mathbf{y})$ so we are done.
Consider $g_1, \cdots g_t$ i.i.d with mean μ, varience σ. Then $\mathbb{E}(\theta_t) = \mathbb{E}(\theta_0) + \sum_{i=1}^{t} v_i$ where $g_i = \mu$, $\theta_t = \theta_{t-1} + \mathbf{v}_t$ with $\mathbb{E}(\theta_0) = \theta_0$, and $\mathbf{v}_t = -\varepsilon \sum_{j=1}^{t} g_j \alpha^{t-j} = -\varepsilon\mu \sum_{j=1}^{t} \alpha^{t-j}$ to get $\mathbb{E}(\theta_t) = \mathbb{E}(\theta_0) + \sum_{i=1}^{t} \sum_{j=1}^{t} \alpha^{t-j}$ where $g_i = \mu$,
Mapping The gradient along the $\mathbf{w}_2$ direction is higher than the gradient along $\mathbf{w}_1$, direction. This is because the contour lines along $\mathbf{w}_2$ directions are very close to each other which indicates a steep curve in the $\mathbf{w}_2$direction. The contour lines along w_1 direction are further apart and hence has slower descent. (or lower gradient)
Infinity norm
$\|\mathbf{x}\| = \max\{|x_i|\}$, $\text{LSE}(\mathbf{x}) = \sum e^{x_i}$. Show $\|\mathbf{x}\|_\infty \leq \text{LSE}(\mathbf{x}) \leq \|x\|_\infty + \ln(n)$. Then $n = 1 \to x \leq x \leq x + 0$. ✓
$n = n+1 \to x \leq \text{LSE}(\mathbf{x}) = \mathbf{x} + \ln(n+1)$ so $e^x \leq \sum_{i=1}^{n+1} e^{x_i} \leq e^{x + \ln(n+1)}$ or
$e^x \leq \sum_{i=1}^{n} e^{x_i} + e^{x_{n+1}} \leq (n+1)e^x = ne^x + e^x$. By the inductive hypothesis, $\sum e^{x_i} \leq ne^x$ but since $e^{x_i} \leq e^x$, we have $e^x \leq \sum_{i=1}^{n} e^{x_i} + e^{x_{n+1}} \leq (n+1)e^x$, so we are done. ✓
To show $\|\mathbf{x}\|_\infty \leq \frac{1}{t}\text{LSE}(t\mathbf{x}) \leq \|x\|_\infty + \frac{1}{t}\ln(n)$, set $\mathbf{x} := t\mathbf{x}$. Then we are done. 5, a, d, e:


(a) Given equation of nesterov momentum

$$\left.\begin{aligned} v_t &= \alpha v_{t-1} - \epsilon \nabla_\theta L(\theta_{t-1} + \alpha v_{t-1}) \\ \theta_t &= \theta_{t-1} + v_t \end{aligned}\right\} \text{——} ①$$

To prove it is equivalent to:

$$\left.\begin{aligned} v_{new} &= \alpha v_{old} - \epsilon \nabla_{\tilde\theta} L(\tilde\theta_{old}) \\ \tilde\theta_{new} &= \tilde\theta_{old} + v_{new} + \alpha(v_{new} - v_{old}) \end{aligned}\right\} \text{——} ②$$

$\Rightarrow$ Let us assume that new parameter $\tilde\theta$

$\tilde\theta = \theta + \alpha V$

This is : $\tilde\theta_{old} = \theta_{old} + \alpha V_{old}$ | $\theta_{old} = \tilde\theta_{old} - \alpha V_{old}$
$\tilde\theta_{new} = \theta_{new} + \alpha V_{new}$ | $\theta_{new} = \tilde\theta_{new} - \alpha V_{new}$

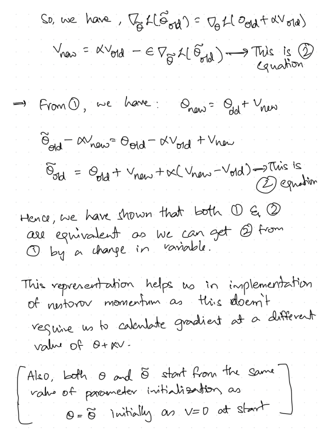
$\Rightarrow$ From ①, we have:

$v_{new} = \alpha V_{old} - \epsilon \nabla_\theta L(\theta_{old} + \alpha V_{old})$

$$\nabla_{\tilde\theta} L(\tilde\theta_{old}) = \frac{\partial L(\tilde\theta_{old})}{\partial\tilde\theta} = \frac{\partial\theta}{\partial\tilde\theta} \cdot \frac{\partial L(\tilde\theta_{old})}{\partial\theta}$$

$$= \overset{\nearrow I}{\cancel{\frac{\partial\theta}{\partial\tilde\theta}}} \cdot \frac{\partial L(\theta_{old} + \alpha V_{old})}{\partial\theta} = \nabla_\theta L(\theta_{old} + \alpha V_{old})$$

$$\left[\because \frac{\partial\theta}{\partial\tilde\theta} = \frac{\partial}{\partial\tilde\theta}(\tilde\theta + \alpha V) = I + 0 = I\right]$$


So, we have , $\nabla_{\tilde\theta} L(\tilde\theta_{old}) = \nabla_\theta L(\theta_{old} + \alpha V_{old})$

$V_{new} = \alpha V_{old} - \epsilon \nabla_{\tilde\theta} L(\tilde\theta_{old}) \longrightarrow$ This is ② equation

$\Rightarrow$ From ①, we have : $\theta_{new} = \theta_{old} + V_{new}$

$\tilde\theta_{old} - \alpha V_{new} = \theta_{old} - \alpha V_{old} + V_{new}$

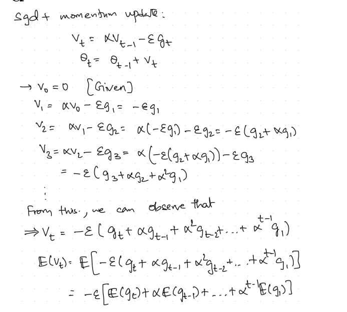
$\tilde\theta_{old} = \theta_{old} + V_{new} + \alpha(V_{new} - V_{old}) \longrightarrow$ This is ② equation

Hence, we have shown that both ① & ② are equivalent as we can get ② from ① by a change in variable.

This representation helps us in implementation of nestorov momentum as this doesn't require us to calculate gradient at a different value of $\theta + \alpha V$.

[Also, both θ and $\tilde\theta$ start from the same value of parameter initialization as $\theta = \tilde\theta$ initially as $V = 0$ at start]

d

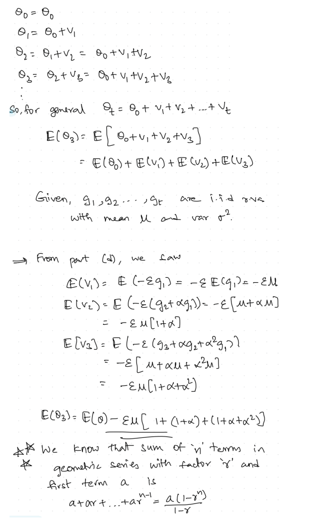

sgd + momentum update:

$V_t = \alpha V_{t-1} - \varepsilon g_t$
$\theta_t = \theta_{t-1} + V_t$

$\rightarrow V_0 = 0$ [Given]
$V_1 = \alpha V_0 - \varepsilon g_1 = -\epsilon g_1$
$V_2 = \alpha V_1 - \varepsilon g_2 = \alpha(-\varepsilon g_1) - \varepsilon g_2 = -\varepsilon(g_2 + \alpha g_1)$
$V_3 = \alpha V_2 - \varepsilon g_3 = \alpha(-\varepsilon(g_2 + \alpha g_1)) - \varepsilon g_3$
$= -\varepsilon(g_3 + \alpha g_2 + \alpha^2 g_1)$
$\vdots$
From this, we can observe that
$\Rightarrow V_t = -\varepsilon(g_t + \alpha g_{t-1} + \alpha^2 g_{t-2} + \ldots + \alpha^{t-1} g_1)$

$\mathbb{E}(V_t) = \mathbb{E}\left[-\varepsilon(g_t + \alpha g_{t-1} + \alpha^2 g_{t-2} + \ldots + \alpha^{t-1} g_1)\right]$
$= -\varepsilon\left[\mathbb{E}(g_t) + \alpha\mathbb{E}(g_{t-1}) + \ldots + \alpha^{t-1}\mathbb{E}(g_1)\right]$

e


$\theta_0 = \theta_0$
$\theta_1 = \theta_0 + V_1$
$\theta_2 = \theta_1 + V_2 = \theta_0 + V_1 + V_2$
$\theta_3 = \theta_2 + V_3 = \theta_0 + V_1 + V_2 + V_3$
$\vdots$
So, for general $\theta_t = \theta_0 + V_1 + V_2 + \ldots + V_t$

$\mathbb{E}(\theta_3) = \mathbb{E}[\theta_0 + V_1 + V_2 + V_3]$
$= \mathbb{E}(\theta_0) + \mathbb{E}(V_1) + \mathbb{E}(V_2) + \mathbb{E}(V_3)$

Given, $g_1, g_2 \ldots, g_t$ are i.i.d r.v.s with mean μ and var σ^2.

$\Rightarrow$ From part (d), we saw
$\mathbb{E}(V_1) = \mathbb{E}(-\varepsilon g_1) = -\varepsilon\,\mathbb{E}(g_1) = -\varepsilon\mu$
$\mathbb{E}(V_2) = \mathbb{E}(-\varepsilon(g_2 + \alpha g_1)) = -\varepsilon[\mu + \alpha\mu]$
$= -\varepsilon\mu[1 + \alpha]$
$\mathbb{E}[V_3] = \mathbb{E}(-\varepsilon(g_3 + \alpha g_2 + \alpha^2 g_1))$
$= -\varepsilon[\mu + \alpha\mu + \alpha^2\mu]$
$= -\varepsilon\mu[1 + \alpha + \alpha^2]$

$\mathbb{E}(\theta_3) = \mathbb{E}(\theta) - \underline{\varepsilon\mu[1 + (1+\alpha) + (1 + \alpha + \alpha^2)]}$

☆☆ We know that sum of 'n' terms in
☆ geometric series with factor 'r' and first term a is

$$a + ar + \ldots + ar^{n-1} = \frac{a(1-r^n)}{1-r}$$

6

a

Suppose,

$$P = \max_i \{ |x_1|, |x_2|, \dots, |x_n| \}$$

Then, we have the following lower bound

$$e^P \leq \sum_{i=1}^{n} e^{|x_i|}$$

We also have the following upper bound

$$\sum_{i=1}^{n} e^{|x_i|} \leq n e^P$$

Combining the upper and lower bounds we have

b

Clearly for $n > 1$,

$$e^P < \sum_{i=1}^{n} e^{|x_i|}$$

Taking natural logarithm of both sides,

$$\ln(e^P) < \ln\left(\sum_{i=1}^{n} e^{|x_i|}\right)$$

$$P\ln(e) < \ln\left(\sum_{i=1}^{n} e^{|x_i|}\right)$$

$$\|X\|_\infty < LSE(X)$$

Discussion

Recall that for $A \in \mathbb{R}^{m \times n}$, it's infinity norm is defined as

$$\|A\|_\infty = \max_i \sum_{j=1}^{n} |a_{ij}|$$

Observe that,

$$\sum_{j=1}^{n} |a_{ij}| \leftarrow$$ Sum of the absolute values of elements in ith row

Hence, infinity norm of a matrix is the maximum absolute row sum.

$$e^P \leq \sum_{i=1}^{n} e^{|x_i|} \leq n e^P$$

Taking natural logarithm of the above Inequality we have

$$\ln(e^P) \leq \ln\left(\sum_{i=1}^{n} e^{|x_i|}\right) \leq \ln(n e^P)$$

$$\Rightarrow P\ln e \leq \ln\left(\sum_{i=1}^{n} e^{|x_i|}\right) \leq \ln(n) + P\ln(e)$$

Since $P = \|X\|_\infty$, so we have the required inequality

$$\|X\|_\infty \leq LSE(X) \leq \|X\|_\infty + \ln(n)$$

c

Suppose,

$$|x_i| = |x_j|$$

for all $i, j \in n$. Then we have

$$\sum_{i=1}^{n} e^{|x_i|}$$

$$= \sum_{i=1}^{n} e^P$$

$$= n e^P$$

tence, the upper bound will e satisfied with equality

$$LSE(X) = \|X\|_\infty + \ln(n)$$

d

) Let $t > 0$ be some scaling constant. Substituting X with tx in (2), we get

$$\|tx\|_\infty \leq LSE(tx) \leq \|tx\|_\infty + \ln(n)$$

Now,

$$\|tx\|_\infty = t\|x\|_\infty$$

So,

$$t\|x\|_\infty \leq LSE(tx) \leq t\|x\|_\infty + \ln(n)$$

Jividing by t, we get the required inequality,

$$\|X\|_\infty \leq \frac{1}{t} LSE(tx) \leq \|X\|_\infty + \frac{\ln(n)}{t}$$

Let

$$A = \begin{pmatrix} 1 & 2 & -4 \\ 3 & 0 & 12 \\ -20 & -1 & 2 \end{pmatrix}$$

Then,

$$\|A\|_\infty = \max\left(\{1+2+4\}, \{3+12\}, \{20+1+2\}\right)$$

$$= 23$$

Now, coming back to our problem

$$W = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nn} \end{bmatrix}$$

$$\frac{\partial \|W\|_\infty}{\partial W} = \begin{bmatrix} \frac{\partial \|W\|_\infty}{\partial w_{11}} & \cdots & \frac{\partial \|W\|_\infty}{\partial w_{1n}} \\ \vdots & & \\ \frac{\partial \|W\|_\infty}{\partial w_{n1}} & \cdots & \frac{\partial \|W\|_\infty}{\partial w_{nn}} \end{bmatrix}$$

Suppose, I tell you that the 3rd row of W has the maximum absolute row sum. Then,

$$\|W\|_\infty = \sum_{j=1}^{n} |w_{3j}|$$

$$= |w_{31}| + |w_{32}| + \dots + |w_{3n}|$$

Then,

$$\frac{\partial \|W\|_\infty}{\partial W} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \text{sign}(w_{31}) & \text{sign}(w_{32}) & \cdots & \text{sign}(w_{3n}) \\ 0 & 0 & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

Then, from the above pattern we can conclude that

$$\frac{\partial \|W\|_\infty}{\partial W} = \begin{bmatrix} - 0 - \\ - 0 - \\ \vdots \\ - \text{sign}(w_K) - \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

where K is the index of the row with maximum absolute sum