Deep Learning

David Zhao

Febrary 9, 2023

1 Preface

This paper is a learning documentation adapted/copied from the book Dive into Deep learning. Coding implementations are omitted (for now).

2 Preliminaries

AutoDifferentiation

Linear Algebra

Norms

Some of the most useful operators in linear algebra are norms. Informally, the norm of a vector tells us how big it is. Here, we are employing a notion of size that concerns the magnitude of a vector's components (not its dimensionality).

A norm is a function $||\cdot||$ that maps a vector to a scalar and satisfies the following three properties:

1. Given any vector $\mathbf{x}$, if we scale the vector by a scalar $\alpha \in \mathbb{R}$, its norm scales accordingly:
$$||\alpha\mathbf{x}|| = |\alpha|||\mathbf{x}||$$
2. For any vectors $\mathbf{x}$ and $\mathbf{y}$, norms must satisfy the triangle inequality:
$$||\mathbf{x} + \mathbf{y}|| = ||\mathbf{x}|| + ||\mathbf{y}||$$
3. The norm of a vector is nonnegative and vanishes if and only if the vector is zero:
$$||\mathbf{x}|| > 0 \iff \mathbf{x} \neq 0$$

For instance, the $l2$ norm measures the (Euclidean) length of a vector which we've all seen already in school when calculating the hypotenuse of a right triangle.. Formally, we know it as

$$||\mathbf{x}||_2 = \sqrt{\sum_{i=1}^{n} \mathbf{x}_i^2}$$

The $l1$ norm is also popular and the associated metric is called the Manhattan distance. By definition, the norm sums the absolute values of a vector's elements:

$$||\mathbf{x}||_1 = \sum_{i=1}^{n} |\mathbf{x}_i|$$

Both the $l1$ and $l2$ norms are special cases of the more general norms:

$$||\mathbf{x}||_p = \left(\sum_{i=1}^{n} \mathbf{x}_i^p \right)^{1/p} \tag{1}$$

Chain Rule

Let $y = f(\mathbf{u})$ such that $\forall i, \mathbf{u}_i = g_i(\mathbf{x})$ where $\mathbf{u} = (u_1, u_2, \ldots, u_m)$ and $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. Then

$$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial u_1} \frac{\partial u_1}{\partial x_i} + \ldots + \frac{\partial y}{\partial u_m} \frac{\partial u_m}{\partial x_i} \tag{2}$$

Baye's Theorem

Given any event A and B such that $P(B) \neq 0$,

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{3}$$

Expectations

3 Linear Neural Networks for Regression

We assume that the relationship between features $\mathbf{x}$ and target y is approximately linear, i.e,

$$E[Y|X = \mathbf{x}] = x_1 w_1 + \ldots + x_d w_d + b \tag{4}$$

where d is the *feature dimensionality*, and b is the *bias*. As such,

$$\hat{y} = \mathbf{w}^T \mathbf{x} + b \quad = \mathbf{X}\mathbf{w} + b \tag{5}$$

In essence, our goal is to find parameters $\mathbf{w}$ and b such that our prediction error is minimized for new data examples that are sampled from the same distribution X.

Loss Function

Naturally, our model requires an objective measure of how well or unwell it fits the training data. Loss functions fill in this role by quantifying the distance between the *observed* and *predicted* labels. The most commonly used loss function is the squared error.

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}(\hat{y}^{(i)} - \mathbf{y}^{(i)})^2 \tag{6}$$

Note that the presence of the constant coefficient $\frac{1}{2}$ is notationally convenient as it disappears when we take the derivative of the loss function. Also notice that large differences

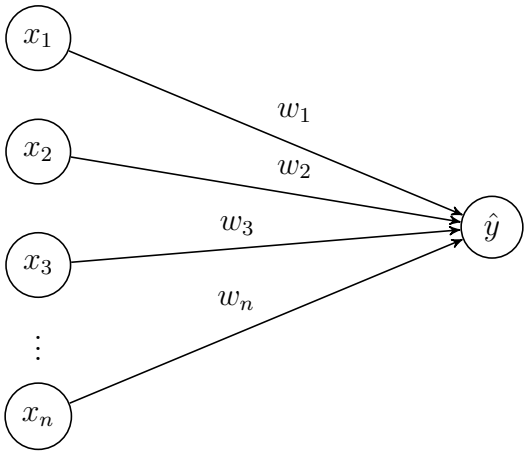

Figure 1: Linear Regression Model with feature dimensionality n

between estimates $\hat{y}^{(i)}$ and targets $\mathbf{y}^{(i)}$ lead to larger contributions due to the function's quadratic form. In fact, while it does encourage our model to avoid sizeable errors, it also yields an excessive sensitivity to anomalous data. To evaluate our model's performance over entire the dataset of n examples, we simply take the average of the losses on the training set:

$$L^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}(\hat{y}^{(i)} - \mathbf{y}^{(i)})^2 \tag{7}$$

Our goal is to find parameters $\mathbf{w}^*$ and b^* that can minizmize the total loss across all examples.

Minibatch Stochastic Gradient Descent

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\beta|}\sum_{i\in\beta_i}\frac{\partial}{\partial(\mathbf{w}, b)}l^{(i)}(\mathbf{w}, b)$$

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{|\beta|}\sum_{i\in\beta_i}\frac{\partial}{\partial\mathbf{w}}l^{(i)}(\mathbf{w}, b) = \mathbf{w} - \frac{\eta}{|\beta|}\sum_{i\in\beta_i}\mathbf{x}^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b - \mathbf{y}^{(i)})$$

$$b \leftarrow b - \frac{\eta}{|\beta|}\sum_{i\in\beta_i}\frac{\partial}{\partial b}l^{(i)}(\mathbf{w}, b) = \mathbf{w} - \frac{\eta}{|\beta|}\sum_{i\in\beta_i}(\mathbf{w}^T\mathbf{x}^{(i)} + b - \mathbf{y}^{(i)})$$

Normal Distribution and Squared Loss

So far we have given a fairly functional motivation of the squared loss objective: the optimal parameters return the conditional expectation whenever the underlying pattern is truly linear, and the loss assigns outsize penalties for outliers. We can also provide a more formal motivation for the squared loss objective by making probabilistic assumptions about the distribution of noise.

To begin, recall that a normal distribution with mean μ and variance σ^2 (standard deviation σ) is given as

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right)$$

One way to motivate linear regression with squared loss is to assume that observations arise from noisy measurements, where the noise is normally distributed as follows:

$$y = \mathbf{w}^T\mathbf{x} + b + \epsilon \quad \text{where } \epsilon \sim \mathcal{N}(0,\, \sigma^2)$$

Thus, we can now write out the likelihood of seeing a particular y for a given $\mathbf{x}$ via

$$P(y|\mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^T\mathbf{x} - b)^2\right)$$

As such, the likelihood factorizes. According to the *principle of maximum likelihood*, the best values of parameters $\mathbf{w}$ and b are those that maximize the likelihood of the entire dataset:

$$P(y|X) = \prod_{i=1}^{n} P(\mathbf{y}^{(i)}|\mathbf{x}^{(i)})$$

since all pairs $(\mathbf{x}^{(i)}, y^{(i)})$ were drawn independently of each other. But, maximizing the product of exponential functions is awkward. Instead, we minimize the negative log-likelihood:

$$\begin{aligned} -\log(y|X) &= -log(\prod_{i=1}^{n} P(\mathbf{y}^{(i)}|\mathbf{x}^{(i)})) \\ &= \sum_{i=1}^{n} \frac{1}{2}\log\big(2\pi\sigma^2\big) + \frac{1}{2\sigma^2}(\mathbf{y}^{(i)} - \mathbf{w}^T\mathbf{x}^{(i)} - b)^2 \end{aligned}$$

It follows that minimizing the square error loss is equivalent to the maximum likelihood estimation of a linear model under additive Gaussian noise.

Generalization

The phenomenon of our model fitting closer to the training model than to the underlying distribution is called *overfitting*. Instead, our goal is to train our model in such a way that it may find a generalizable pattern and make correct predictions about previously unseen data.

Training Error & Generalization Error

In standard supervised learning setting, we assume the training and testing data to be drawn independently from identical distributions (i.e. *IID* assumption). Training error (R_{emp}) is a statistic calculated on the training dataset:

$$R_{emp}[X, \mathbf{y}, f] = \frac{1}{n}\sum_{i=1}^{n} l(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, f(\mathbf{x})^{(i)})$$

Generalization error (R) is an expectation taken with respect to the underlying distribution:

$$R[p, f] = E_{(\mathbf{x},y)\sim P[l(\mathbf{x},y,f(\mathbf{x})]} = \int\int l(\mathbf{x}, y, f(\mathbf{x}))p(\mathbf{x}, y)d\mathbf{x}dy$$

Note that we can never measure R exactly since the density function $p(\mathbf{x}, y)$ has a form that can almost never be precisely known. Moreover, since we cannot sample an infinite stream of data points, we must resort to estimating the generalization error by applying our model to an independent test set that is withheld from our training set.

Model Complexity

Intuitively, when we have simple models mixed with abundant data, the training and generalization error tend to be close. Conversely, we can expect more a complex model and/or fewer examples to cause our training error to diminish, but the generalization error to grow. Error on the holdout data, i.e. the validation set, is called the *validation error*.

Polynomial Curve Fitting

Cross Validation

In cases when we are dealt with scarce training data, it is likely that we often lack enough hold out data to form a validation set. A popular solution is to use *K-fold cross-validation* where the training data is first partitioned into k disjoints sets. Then, we perform a total of k training/validation steps, each time training on $k-1$ sets and validating on the remaining unused set. Finally, we average the training and validation errors over the results obtained from our k experiments.

Weight Decay

Recall that we can always mitigate overfitting by collecting more training data. However, gathering more data is often costly, time consuming, etc. Therefore, we introduce our first *regularization* technique known as *weight decay*.

Note that we may also limit model complexity by tweaking the degree of our fitted polynomial. However, even small changes in degree can dramatically increase model complexity, hence motivating our necessity for a more fine-tuning method, i.e. weight decay.

Norms & Weight Decay

$$\mathbf{w} \leftarrow (1-\eta\lambda)\mathbf{w} - \frac{\eta}{|\beta|}\sum_{i\in\beta_i}\frac{\partial}{\partial \mathbf{w}} l^{(i)}(\mathbf{w}, b) = \mathbf{w} - \frac{\eta}{|\beta|}\sum_{i\in\beta_i}\mathbf{x}^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b - \mathbf{y}^{(i)})$$

4 Linear Neural Networks for Classification

Classification

To get our feet wet, let's start with a simple image classification problem. Here, each input consists of a 2×2 grayscale image. We can represent each pixel value with a single scalar, giving us four features x_1, x_2, x_3, x_4. Further, let's assume that each image belongs to one among the categories "cat", "chicken", and "dog".

In general, classification problems do not come with natural orderings among the classes. Fortunately, statisticians long ago invented a simple way to represent categorical data: the *one-hot encoding*. A one-hot encoding is a vector with as many components as we have categories. The component corresponding to a particular instance's category is set to 1 and all other components are set to 0. In our case, a label would be a three-dimensional vector, with $(1,0,0)$ corresponding to "cat", $(0,1,0)$ to "chicken", and $(0,0,1)$ to "dog".

Linear Model

$$o_1 = x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1,$$
$$o_2 = x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2,$$
$$o_3 = x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3.$$

Assuming a suitable loss function, we could try, directly, to minimize the difference between and the labels . While it turns out that treating classification as a vector-valued regression problem works surprisingly well, it is nonetheless lacking in the following ways:

- There is no guarantee that the outputs σ_i sum up to 1 in the way we expect probabilities to behave.
- There is no guarantee that the outputs σ_i are even nonnegative, even if their outputs sum up to 1, or that they do not exceed 1

Softmax

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}.$$

Log-Likelihood

$$P(\mathbf{Y} \mid \mathbf{X}) = \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}).$$

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j \log \hat{y}_j$$

Softmax and Cross-Entropy Loss

$$\begin{aligned} l(\mathbf{y}, \hat{\mathbf{y}}) &= -\sum_{j=1}^{q} y_j \log \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} \\ &= \sum_{j=1}^{q} y_j \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j \\ &= \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j. \end{aligned}$$

$$\partial_{o_j} l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} - y_j = \text{softmax}(\mathbf{o})_j - y_j.$$

5 Multilayer Perceptrons

Recall in section 3, we described affine transformations as linear transformations with added bias. This model maps inputs directly to outputs via a single affine transformation, followed by a softmax operation. However, linearity is often a strong assumption.

Limitations of Linear Models

Linearity implies the weaker law of *monotonicity* i.e. any increase in inputs must always correspond to an increase in our model's output (positive weights), or a decrease in our model's output (negative weights). Often times, linearity becomes too strong of an assumption to be applied to problems that require more specific modelling. Suppose for example we want to predict whether an individual will repay loan based on their salary. Although this relationship is monotonic, it is perhaps not linear as an increase in income from \$0 to \$50,000 likely corresponds to a higher likelihood of repayment than an increase from \$1 million to \$1.05 million. As such, it may be preferable to post-process our outcome by using a logarithmic map, to make linearity a more plausible assumption.

Incorporating Hidden Layers

We overcome the limitations of linearity by incorporating one or more hidden layers. The most common way to do this is to stack many fully connected layers on top of each other. We can think of the first $L-1$ layers as our representation and the final layer as our linear predictor. This architecture is commonly called a *multilayer preceptron* or *MLP*.

From Linear to Nonlinear

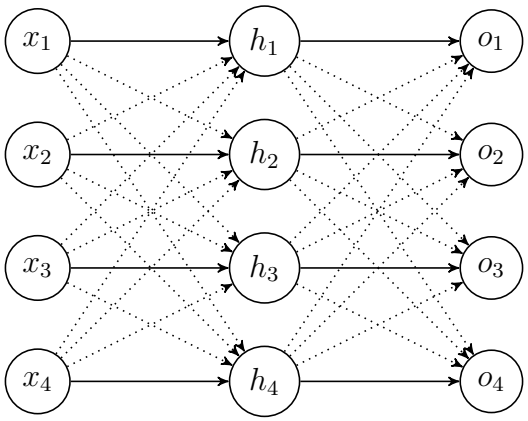

Figure 2: An MLP with a hidden layer of 4 hidden units.

Activation Functions

ReLU Function

The most popular choice, due to both simplicity of implementation and its good performance on a variety of predictive tasks, is the rectified linear unit (ReLU) ReLU provides a very simple nonlinear transformation. Given an element x, the function is defined as the maximum of that element and 0:

$$ReLU(x) = \max(x, 0)$$

When the input is negative, the derivative of the ReLU function is 0, and when the input is positive, the derivative of the ReLU function is 1. Note that the ReLU function is not differentiable when the input takes value precisely equal to 0. In these cases, we default to the left-hand-side derivative and say that the derivative is 0 when the input is 0.

Sigmoid Function

$$sigmoid(x) = \frac{1}{1 + \exp(-x)}$$

Tanh Function

$$tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$

Forward and Backward Propagation, Computational Graphs

Forward Propagation

Forward propagation (or forward pass) refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer.

Backward Propagation

Numerical Stability and Initialization

Vanishing and Exploding Gradients

Parameter Initialization

One way of addressing—or at least mitigating—the issues raised above is through careful initialization. As we will see later, additional care during optimization and suitable regularization can further enhance stability.

Default Initialization

In the previous sections, we used a normal distribution to initialize the values of our weights. If we do not specify the initialization method, the framework will use a default random initialization method, which often works well in practice for moderate problem sizes.

Generalization in Deep Learning

The TL;DR of the present moment is that the theory of deep learning has produced promising lines of attack and scattered fascinating results, but still appears far from a comprehensive account of both (i) why we are able to optimize neural networks and (ii) how models learned by gradient descent manage to generalize so well, even on high-dimensional tasks. However, in practice, (i) is seldom a problem (we can always find parameters that will fit all of our training data) and thus understanding generalization is far the bigger problem. On the other hand, even absent the comfort of a coherent scientific theory, practitioners have developed a large collection of techniques that may help you to produce models that generalize well in practice. While no pithy summary can possibly do justice to the vast topic of generalization in deep learning, and while the overall state of research is far from resolved, we hope, in this section, to present a broad overview of the state of research and practice.

Dropout

6 Builder's Guide

7 Convolutional Neural Networks

Image data is represented as a two-dimensional grid of pixels, be it monochromatic or in color. Accordingly each pixel corresponds to one or multiple numerical values respectively. So far we ignored this rich structure and treated them as vectors of numbers by flattening the images, irrespective of the spatial relation between pixels. This deeply unsatisfying approach was necessary in order to feed the resulting one-dimensional vectors through a fully connected MLP.

Because these networks are invariant to the order of the features, we could get similar results regardless of whether we preserve an order corresponding to the spatial structure of the pixels or if we permute the columns of our design matrix before fitting the MLP's parameters. Preferably, we would leverage our prior knowledge that nearby pixels are typically related to each other, to build efficient models for learning from image data.

From Fully Connected Layers to Convolutions

To start off, we can consider an MLP with two-dimensional images $\mathbf{X}$ as inputs and their immediate hidden representations $\mathbf{H}$ similarly represented as matrices (they are two-dimensional tensors in code), where both $\mathbf{X}$ and $\mathbf{H}$ have the same shape. Let that sink in. We now conceive of not only the inputs but also the hidden representations as possessing spatial structure.

$$\begin{aligned}[\mathbf{H}]_{i,j} &= [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathsf{W}]_{i,j,k,l}[\mathbf{X}]_{k,l} \\ &= [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathsf{V}]_{i,j,a,b}[\mathbf{X}]_{i+a,j+b}.\end{aligned}$$

Translation Invariance

$$[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{V}]_{a,b}[\mathbf{X}]_{i+a,j+b}.$$

Spatial Locality

We believe that we should not have to look very far away from location (i, j) in order to glean relevant information to assess what is going on at $[\mathbf{H}]_{i,j}$. This means that outside some range $|a| > \Delta$ or $|b| > \Delta$, we should set $[\mathbf{V}]_{a,b} = 0$. Equivalently, we can rewrite $[\mathbf{H}]_{i,j}$ as

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b}[\mathbf{X}]_{i+a,j+b}. \tag{8}$$

where Δ is typically smaller than 10. Note that (7.1.3), in a nutshell, is what is called a convolutional layer. Convolutional neural networks (CNNs) are a special family of neural networks that contain convolutional layers. In the deep learning research community, $\mathbf{V}$ is

referred to as a *convolution kernel*, a *filter*, or simply the layer's *weights* that are learnable parameters.

Convolutions

Let's briefly review why (7.1.3) is called a convolution. In mathematics, the convolution between two functions (Rudin, 1973), say is defined as

$$(f * g)(\mathbf{x}) = \int f(\mathbf{z})g(\mathbf{x} - \mathbf{z})d\mathbf{z}.$$

$$(f * g)(i) = \sum_a f(a)g(i - a).$$

$$(f * g)(i, j) = \sum_a \sum_b f(a, b)g(i - a, j - b).$$

Channels

$$[\mathsf{H}]_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_c [\mathsf{V}]_{a,b,c,d}[\mathsf{X}]_{i+a,j+b,c},$$

Cross-Correlation Operation

$$(n_h - k_h + 1) \times (n_w - k_w + 1)$$

Padding

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1).$$

Stride

Pooling

Like convolutional layers, *pooling* operators consist of a fixed-shape window that is slid over all regions in the input according to its stride, computing a single output for each location traversed by the fixed-shape window (sometimes known as the *pooling window*). However, unlike the cross-correlation computation of the inputs and kernels in the convolutional layer, the pooling layer contains no parameters (there is no *kernel*). Instead, pooling operators are deterministic, typically calculating either the maximum or the average value of the elements in the pooling window. These operations are called *maximum pooling* and *average pooling*, respectively.

8 Modern Convolutional Neural Networks

Batch Normalization

Denote by $\mathcal{B}$ a minibatch and let $\mathbf{x} \in \mathcal{B}$ be an input to batch normalization (BN). In this case the batch normalization is defined as follows:

$$\mathrm{BN}(\mathbf{x}) = \boldsymbol{\gamma} \odot \frac{\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\hat{\boldsymbol{\sigma}}_{\mathcal{B}}} + \boldsymbol{\beta}.$$

where $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ is the sample mean and $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$ is the sample standard deviation of the minibatch $\mathcal{B}$. After applying standardization, the resulting minibatch has zero mean and unit variance. The choice of unit variance (vs. some other magic number) is an arbitrary choice. We recover this degree of freedom by including an elementwise scale parameter $\boldsymbol{\gamma}$ and shift parameter $\boldsymbol{\beta}$ that have the same shape as $\mathbf{x}$. Both are parameters that need to be learned as part of model training.

$$\hat{\boldsymbol{\mu}}_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x} \text{ and } \hat{\boldsymbol{\sigma}}_{\mathcal{B}}^2 = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}})^2 + \epsilon.$$

Note that we add a small constant $\epsilon > 0$ to the variance estimate to ensure that we never attempt division by zero, even in cases where the empirical variance estimate might be very small or even vanish. The estimates $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ and $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$ counteract the scaling issue by using noisy estimates of mean and variance. You might think that this noisiness should be a problem. Quite to the contrary, this is actually beneficial.

For reasons that are not yet well-characterized theoretically, various sources of noise in optimization often lead to faster training and less overfitting: batch normalization works best for moderate minibatches sizes in the $50 \sim 100$ range. This particular size of minibatch seems to inject just the "right amount" of noise per layer, both in terms of scale via $\hat{\boldsymbol{\sigma}}$, and in terms of offset via $\hat{\boldsymbol{\mu}}$: a larger minibatch regularizes less due to the more stable estimates, whereas tiny minibatches destroy useful signal due to high variance.

Layer Normalization

Fully Connected Layers

$$\mathbf{h} = \phi(\mathrm{BN}(\mathbf{W}\mathbf{x} + \mathbf{b})).$$

Convolutional Layers

Residual Networks (ResNet) and ResNeXt

9 Recurrent Neural Networks

10 Modern Recurrent Neural Networks

11 Attention Mechanisms and Transformers

Queries, Keys, and Values

Denote by $\mathcal{D} \stackrel{\text{def}}{=} \{(\mathbf{k}_1, \mathbf{v}_1), \ldots (\mathbf{k}_m, \mathbf{v}_m)\}$ a database of m tuples of *keys* and *values*. Moreover, denote by $\mathbf{q}$ a *query*. Then we can define the *attention* over $\mathcal{D}$

$$\mathrm{Attention}(\mathbf{q}, \mathcal{D}) \stackrel{\text{def}}{=} \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i) \mathbf{v}_i,$$

where $\alpha(\mathbf{q}, \mathbf{k}_i) \in \mathbb{R}$ are scalar attention weights. The operation itself is typically referred to as *attention pooling*. The name attention derives from the fact that the operation pays particular attention to the terms for which the weight α is significant (i.e., large). As such,

the attention over $\mathcal{D}$ generates a linear combination of values contained in the database. This opens a number of special cases, namely

- The weights $\alpha(\mathbf{q}, \mathbf{k}_i)$ are nonnegative. In this case the output of the attention mechanism is contained in the convex cone spanned by the values $\mathbf{v}_i$.
- The weights $\alpha(\mathbf{q}, \mathbf{k}_i)$ form a convex combination, i.e., $\sum_i \alpha(\mathbf{q}, \mathbf{k}_i) = 1$ and $\alpha(\mathbf{q}, \mathbf{k}_i) \geq 0$ for all i. This is the most common setting in deep learning.
- Exactly one of the weights $\alpha(\mathbf{q}, \mathbf{k}_i)$ is 1, while all others are 0. This is akin to a traditional database query.
- All weights are equal, i.e., $\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{1}{m}$ for all i. This amounts to averaging across the entire database, also called average pooling in deep learning.

A common strategy to ensure that the weights sum up to 1 and that they are also nonnegative is to apply the softmax operation used for multinomial models via

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_j \exp(a(\mathbf{q}, \mathbf{k}_j))}.$$

Attention Pooling by Similarity

Now that we introduced the primary components of the attention mechanism, let's use them in a rather classical setting, namely regression and classification via kernel density estimation (Nadaraya, 1964, Watson, 1964). At their core, Nadaraya-Watson estimators rely on some similarity kernel $\alpha(\mathbf{q}, \mathbf{k})$ relating queries $\mathbf{q}$ to keys $\mathbf{k}$. Some common kernels are

$$\begin{aligned}
\alpha(\mathbf{q}, \mathbf{k}) &= \exp\left(-\frac{1}{2}\|\mathbf{q} - \mathbf{k}\|^2\right) && \text{Gaussian} \\
\alpha(\mathbf{q}, \mathbf{k}) &= 1 \text{ if } \|\mathbf{q} - \mathbf{k}\| \leq 1 && \text{Boxcar} \\
\alpha(\mathbf{q}, \mathbf{k}) &= \max\left(0, 1 - \|\mathbf{q} - \mathbf{k}\|\right) && \text{Epanechikov}
\end{aligned}$$

All of the kernels are heuristic and can be tuned. For instance, we can adjust the width, not only on a global basis but even on a per-coordinate basis. Regardless, all of them lead to the following equation for regression and classification alike:

$$f(\mathbf{q}) = \sum_i \mathbf{v}_i \frac{\alpha(\mathbf{q}, \mathbf{k}_i)}{\sum_j \alpha(\mathbf{q}, \mathbf{k}_j)}.$$

In the case of a (scalar) regression with observations $(\mathbf{x}_i, y_i)$ for features and labels respectively, $\mathbf{v}_i = y_i$ are scalars, $\mathbf{k}_i = \mathbf{x}_i$ are vectors, and the query $\mathbf{q}$ denotes the new location where f should be evaluated. In the case of (multiclass) classification, we use one-hot-encoding of y_i to obtain $\mathbf{v}_i$. One of the convenient properties of this estimator is that it requires no training. Even more so, if we suitably narrow the kernel with increasing amounts of data, the approach is consistent (Mack and Silverman, 1982), i.e., it will converge to some statistically optimal solution.

Attention Scoring Functions

As it turns out, distance functions are slightly more expensive to compute than inner products. As such, with the softmax operation to ensure nonnegative attention weights, much of the work has gone into attention scoring functions α that are simpler to compute.

Dot Product Attention

Let's review the attention function (without exponentiation) from the Gaussian kernel for a moment:

$$\begin{aligned} a(\mathbf{q}, \mathbf{k}_i) &= -\frac{1}{2}\|\mathbf{q} - \mathbf{k}_i\|^2 \\ &= \mathbf{q}^\top \mathbf{k}_i - \frac{1}{2}\|\mathbf{k}_i\|^2 - \frac{1}{2}\|\mathbf{q}\|^2. \end{aligned}$$

Note that the last term depends on $\mathbf{q}$ only, so it is identical for all (q, k_i) pairs. Normalizing the attention weights to 1 ensures that this term disappears completely. Also note that batch and layer normalization lead to activations that have well-bounded, and often constant norms $\|\mathbf{k}_i\| \approx \text{const}$. This is the case, for instance, whenever the keys $\mathbf{k}_i$ were generated by a layer norm. As such, we can drop it from the definition of a without any major change in the outcome.

Last, we need to keep the order of magnitude of the arguments in the exponential function under control. Assume that all the elements of the query $\mathbf{q} \in \mathbb{R}^d$ and the key $\mathbf{k}_i \in \mathbb{R}^d$ are independent and identically drawn random variables with zero mean and unit variance. The dot product between both vectors has zero mean and a variance of d. To ensure that the variance of the dot product still remains one regardless of vector length, we use the *scaled dot-product attention* scoring function. That is, we rescale the dot-product by $1/\sqrt{d}$. We thus arrive at the first commonly used attention function that is used, e.g., in Transformers

$$a(\mathbf{q}, \mathbf{k}_i) = \mathbf{q}^\top \mathbf{k}_i / \sqrt{d}.$$

Note that attention weights α still need normalizing. We can simplify this further by using the softmax operation:

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \text{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp\left(\mathbf{q}^\top \mathbf{k}_i / \sqrt{d}\right)}{\sum_{j=1} \exp\left(\mathbf{q}^\top \mathbf{k}_j / \sqrt{d}\right)}.$$

Masked Softmax Operation

One of the most popular applications of the attention mechanism is to sequence models. Hence we need to be able to deal with sequences of different lengths. In some cases, such sequences may end up in the same minibatch, necessitating padding with dummy tokens for shorter sequences

Since we do not want blanks in our attention model we simply need to limit $\sum_{i=1}^{n} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i$ to $\sum_{i=1}^{l} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i$ for however long $l \leq n$ the actual sentence is. Since it is such a common problem, it has a name: the *masked softmax operation*.

To actually implement it, we cheat ever so slightly by setting the values to zero $\mathbf{v}_i = 0$ for $i > l$. Moreover, we set the attention weights to a large negative number, such as -10^6 in order to make their contribution to gradients and values vanish in practice (Recall that the softmax operation computes the exponentiation of its input values). This is done since linear algebra kernels and operators are heavily optimized for GPUs and it is faster to be slightly wasteful in computation rather than to have code with conditional statements.

Batch Matrix Multiplication

Another commonly used operation is to multiply batches of matrices with another. This comes in handy when we have minibatches of queries, keys, and values. More specifically,

assume that

$$\mathbf{Q} = [\mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_n] \in \mathbb{R}^{n \times a \times b}$$
$$\mathbf{K} = [\mathbf{K}_1, \mathbf{K}_2, \ldots, \mathbf{K}_n] \in \mathbb{R}^{n \times b \times c}$$

Then the batch matrix multiplication (BMM) computes the element-wise product

$$\text{BMM}(\mathbf{Q}, \mathbf{K}) = [\mathbf{Q}_1\mathbf{K}_1, \mathbf{Q}_2\mathbf{K}_2, \ldots, \mathbf{Q}_n\mathbf{K}_n] \in \mathbb{R}^{n \times a \times c}.$$

Scaled Dot-Product Attention

In practice, we often think in minibatches for efficiency, such as computing attention for n queries and m key-value pairs, where queries and keys are of length d and values are of length v. The scaled dot-product attention of queries $\mathbf{Q} \in \mathbb{R}^{n \times d}$, keys $\mathbf{K} \in \mathbb{R}^{m \times d}$, and values $\mathbf{V} \in \mathbb{R}^{m \times v}$ thus can be written as

$$\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V} \in \mathbb{R}^{n \times v}.$$

Additive Attention

When queries $\mathbf{q}$ and keys $\mathbf{k}$ are vectors of different dimensionalities, we can either use a matrix to address the mismatch via $\mathbf{q}^\top\mathbf{M}\mathbf{k}$, or we can use additive attention as the scoring function. Another benefit is that, as its name indicates, the attention is additive. This can lead to some minor computational savings. Given a query $\mathbf{q} \in \mathbb{R}^q$ and a key $\mathbf{k} \in \mathbb{R}^k$, the additive attention scoring function is given by

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q\mathbf{q} + \mathbf{W}_k\mathbf{k}) \in \mathbb{R},$$

where $\mathbf{W}_q \in \mathbb{R}^{h \times q}$, $\mathbf{W}_k \in \mathbb{R}^{h \times k}$, and $\mathbf{w}_v \in \mathbb{R}^h$ are the learnable parameters. This term is then fed into a softmax to ensure both nonnegativity and normalization. An equivalent interpretation of is that the query and key are concatenated and fed into an MLP with a single hidden layer using tanh as the activation function and disabling bias terms.

The Bahdanau Attention Mechanism

12 Optimization Algorithms

Convexity

Before convex analysis, we need to define *convex* sets and *convex* functions. They lead to mathematical tools that are commonly applied to machine learning.

Convex Sets

$$\lambda a + (1 - \lambda)b \in \mathcal{X} \text{ whenever } a, b \in \mathcal{X}.$$

Convex Functions

Now that we have convex sets we can introduce *convex* functions f . Given a *convex* set $\mathcal{X}$, a function $f : \mathcal{X} \to \mathbb{R}$ is *convex* if for all $x, x' \in \mathcal{X}$ and for all $\lambda \in [0, 1]$ we have

$$\lambda f(x) + (1 - \lambda)f(x') \geq f(\lambda x + (1 - \lambda)x').$$

Jensen's Inequality

Given a convex function f, one of the most useful mathematical tools is *Jensen's inequality*. It amounts to a generalization of the definition of convexity:

$$\sum_i \alpha_i f(x_i) \geq f\left(\sum_i \alpha_i x_i\right) \text{ and } E_X[f(X)] \geq f\left(E_X[X]\right),$$

where α_i are nonnegative real numbers such that $\sum_i \alpha_i = 1$ and X is a random variable. In other words, the expectation of a convex function is no less than the convex function of an expectation, where the latter is usually a simpler expression. To prove the first inequality we repeatedly apply the definition of convexity to one term in the sum at a time.

One of the common applications of Jensen's inequality is to bound a more complicated expression by a simpler one. For example, its application can be with regard to the log-likelihood of partially observed random variables. That is, we use

$$E_{Y \sim P(Y)}[-\log P(X \mid Y)] \geq -\log P(X),$$

Miscellaneous

13 Appendix: Mathematics for Deep Learning

13.1 Maximum Likelihood