Training Convolutional Neural Networks

Carlo Tomasi

November 26, 2018

1 The Soft-Max Simplex

Neural networks are typically designed to compute real-valued functions $\mathbf{y} = h(\mathbf{x}) \ : \ \mathbb{R}^d \to \mathbb{R}^e$ of their input $\mathbf{x}$. When a classifier is needed, a soft-max function is used as the last layer, with e entries in its output vector $\mathbf{p}$ if there are e classes in the label space Y. The class corresponding to input $\mathbf{x}$ is then is found as the $\arg\max$ of $\mathbf{p}$. Thus, the network can be viewed as a function

$$\mathbf{p} = f(\mathbf{x}, \mathbf{w}) \ : \ X \to P$$

that transforms data space X into the *soft-max simplex* P, the set of all nonnegative real-valued vectors $\mathbf{p} \in \mathbb{R}^e$ whose entries add up to 1:

$$P \stackrel{\text{def}}{=} \{\mathbf{p} \in \mathbb{R}^e \ : \ \mathbf{p} \geq \mathbf{0} \ \text{ and } \ \sum_{i=1}^{e} p_i = 1\} \ .$$

This set has dimension $e - 1$, and is the convex hull of the e columns of the identity matrix in $\mathbb{R}^e$. Figure 1 shows the 1-simplex and the 2-simplex.[1]

The vector $\mathbf{w}$ in the expression above collects all the parameters of the neural network, that is, the gains and biases of all the neurons. More specifically, for a deep neural network with K layers indexed by $k = 1, \ldots, K$, we can write

$$\mathbf{w} = \begin{bmatrix} \mathbf{w}^{(1)} \\ \vdots \\ \mathbf{w}^{(K)} \end{bmatrix}$$

where $\mathbf{w}^{(k)}$ is a vector collecting both gains and biases for layer k.

If the $\arg\max$ rule is used to compute the class,

$$\hat{y} = h(\mathbf{x}) = \arg\max \mathbf{p} \ ,$$

then the network has a low training risk if the transformed data points $\mathbf{p}$ fall in the decision regions

$$P_c = \{p_c \geq p_j \ \text{ for } \ j \neq c\} \quad \text{for} \quad c = 1, \ldots, e \ .$$

These regions are convex, because their boundaries are defined by linear inequalities in the entries of $\mathbf{p}$. Thus, when used for classification, the neural network can be viewed as learning a transformation of the original decision regions in X into the convex decision regions in the soft-max simplex.

[1] In geometry, the simplices are named by their dimension, which is one less than the number of classes.

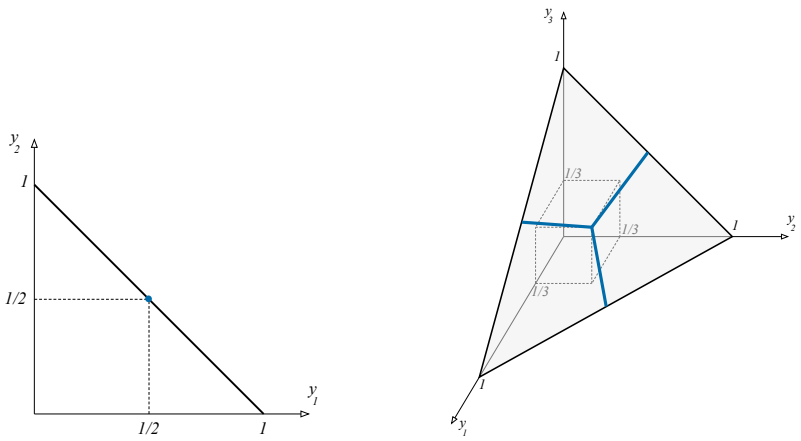

Figure 1: The 1-simplex for two classes (dark segment in the diagram on the left) and the 2-simplex for three classes (light triangle in the diagram on the right). The blue dot on the left and the blue line segments on the right are the boundaries of the decision regions. The boundaries meet at the *unit point* $\mathbf{1}/e$ in e dimensions.

2 Loss

The risk L_T to be minimized to train a neural network is the average loss on a training set of input-output pairs

$$T = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_N, \mathbf{y}_N)\} \ .$$

The outputs $\mathbf{y}_n$ are categorical in a classification problem, and real-valued vectors in a regression problem.

For a regression problem, the loss function is typically the quadratic loss,

$$\ell(\mathbf{y}, \mathbf{y}') = \|\mathbf{y} - \mathbf{y}'\|^2 \ .$$

For classification, on the other hand, we would like the risk $L_T(h)$ to be differentiable, in order to be able to use gradient descent methods. However, the $\arg\max$ is a piecewise-constant function, and its derivatives are either zero or undefined (where the $\arg\max$ changes value). The zero-one loss function has similar properties. To address these issue, a differentiable loss defined on f is used as a proxy for the zero-one loss defined on h. Specifically, the multi-class cross-entropy loss is used, which we studied in the context of logistic-regression classifiers. Its definition is repeated here for convenience:

$$\ell(y, \mathbf{p}) = -\log p_y \ .$$

Equivalently, if $\mathbf{q}(y)$ is the one-hot encoding of the true label y, the cross-entropy loss can also be written as follows:

$$\ell(y, \mathbf{p}) = -\sum_{k=1}^{e} q_i(y) \log p_i \ .$$

With these definitions, L_T, is a piecewise-differentiable function, and one can use gradient or sub-gradient methods to compute the gradient of L_T with respect to the parameter vector $\mathbf{w}$.

Exceptions to differentiability are due to the use of the ReLU, which has a cusp at the origin, as the nonlinearity in neurons, as well as to the possible use of max-pooling. These exceptions are pointwise, and

are typically ignored in both the literature and the software packages used to minimize L_T. If desired, they could be addresses by either computing sub-gradients rather than gradients [3], or rounding out the cusps with differentiable joints.

As usual, once the loss has been settled on, the training risk is defined as the average loss over the training set, and expressed as a function of the parameters $\mathbf{w}$ of f:

$$L_T(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \ell_n(\mathbf{w}) \quad \text{where} \quad \ell_n(\mathbf{w}) = \ell(y_n, f(\mathbf{x}_n, \mathbf{w})) \; . \tag{1}$$

3 Back-Propagation

A local minimum for the risk $L_T(\mathbf{w})$ is found by an iterative procedure that starts with some *initial values* $\mathbf{w}_0$ for $\mathbf{w}$, and then at step t performs the following operations:

- Compute the gradient of the training risk,

$$\left.\frac{\partial L_T}{\partial \mathbf{w}}\right|_{\mathbf{w}=\mathbf{w}_{t-1}} .$$

- Take a step that reduces the value of L_T by moving in the direction of the negative gradient by a variant of the steepest descent method called *Stochastic Gradient Descent* (SGD), discussed in Section 4.

The gradient computation is called *back-propagation* and is described next.

The computation of the n-th loss term $\ell_n(\mathbf{w})$ can be rewritten as follows:

$$\begin{aligned}
\mathbf{x}^{(0)} &= \mathbf{x}_n \\
\mathbf{x}^{(k)} &= f^{(k)}(W^{(k)}\tilde{\mathbf{x}}^{(k-1)}) \quad \text{for} \quad k = 1, \ldots, K \\
\mathbf{p} &= \mathbf{x}^{(K)} \\
\ell_n &= \ell(y_n, \mathbf{p})
\end{aligned}$$

where $(\mathbf{x}_n, y_n)$ is the n-th training sample and $f^{(k)}$ describes the function implemented by layer k.

Computation of the derivatives of the loss term $\ell_n(\mathbf{w})$ can be understood with reference to Figure 2. The term ℓ_n depends on the parameter vector $\mathbf{w}^{(k)}$ for layer k through the output $\mathbf{x}^{(k)}$ from that layer and nothing else, so that we can write

$$\frac{\partial \ell_n}{\partial \mathbf{w}^{(k)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(k)}} \frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{w}^{(k)}} \quad \text{for} \quad k = K, \ldots, 1 \tag{2}$$

and the first gradient on the right-hand side satisfies the backward recursion

$$\frac{\partial \ell_n}{\partial \mathbf{x}^{(k-1)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(k)}} \frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{x}^{(k-1)}} \quad \text{for} \quad k = K, \ldots, 2 \tag{3}$$

because ℓ_n depends on the output $\mathbf{x}^{(k-1)}$ from layer $k-1$ only through the output $\mathbf{x}^{(k)}$ from layer k. The recursion (3) starts with

$$\frac{\partial \ell_n}{\partial \mathbf{x}^{(K)}} = \frac{\partial \ell}{\partial \mathbf{p}} \tag{4}$$

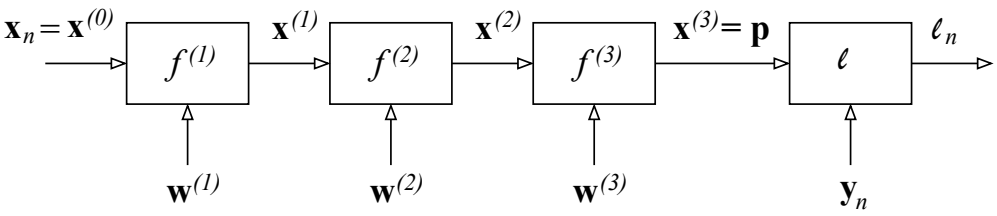

Figure 2: Example data flow for the computation of the loss term ℓ_n for a neural network with $K = 3$ layers. When viewed from the loss term ℓ_n, the output $\mathbf{x}^{(k)}$ from layer k (pick for instance $k = 2$) is a bottleneck of information for both the parameter vector $\mathbf{w}^{(k)}$ for that layer and the output $\mathbf{x}^{(k-1)}$ from the previous layer ($k - 1 = 1$ in the example). This observation justifies the use of the chain rule for differentiation to obtain equations (2) and (3).

where $\mathbf{p}$ is the second argument to the loss function ℓ.

In the equations above, the derivative of a function with respect to a vector is to be interpreted as the *row* vector of all derivatives. Let d_k be the dimensionality (number of entries) of $\mathbf{x}^{(k)}$, and j_k be the dimensionality of $\mathbf{w}^{(k)}$. The two matrices

$$\frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{w}^{(k)}} = \begin{bmatrix} \frac{\partial x_1^{(k)}}{\partial w_1^{(k)}} & \cdots & \frac{\partial x_1^{(k)}}{\partial w_{j_k}^{(k)}} \\ \vdots & & \vdots \\ \frac{\partial x_{d_k}^{(k)}}{\partial w_1^{(k)}} & \cdots & \frac{\partial x_{d_k}^{(k)}}{\partial w_{j_k}^{(k)}} \end{bmatrix} \quad \text{and} \quad \frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{x}^{(k-1)}} = \begin{bmatrix} \frac{\partial x_1^{(k)}}{\partial x_1^{(k-1)}} & \cdots & \frac{\partial x_1^{(k)}}{\partial x_{d_{k-1}}^{(k-1)}} \\ \vdots & & \vdots \\ \frac{\partial x_{d_k}^{(k)}}{\partial x_1^{(k-1)}} & \cdots & \frac{\partial x_{d_k}^{(k)}}{\partial x_{d_{k-1}}^{(k-1)}} \end{bmatrix} \tag{5}$$

are the *Jacobian matrices* of the layer output $\mathbf{x}^{(k)}$ with respect to the layer parameters and inputs. Computation of the entries of these Jacobians is a simple exercise in differentiation, and is left to the Appendix.

The equations (2-5) are the basis for the *back-propagation* algorithm for the computation of the gradient of the training risk $L_T(\mathbf{w})$ with respect to the parameter vector $\mathbf{w}$ of the neural network (Algorithm 1). The algorithm loops over the training samples. For each sample, it feeds the input $\mathbf{x}_n$ to the network to compute the layer outputs $\mathbf{x}^{(k)}$ for that sample and for all $k = 1, \ldots, K$, in this order. The algorithm temporarily stores all the values $\mathbf{x}^{(k)}$, because they are needed to compute the required derivatives. This initial volley of computation is called *forward propagation* (of the inputs). The algorithm then revisits the layers in reverse order while computing the derivatives in equation (4) first and then in equations (2) and (3), and concatenates the resulting K layer gradients into a single gradient $\frac{\partial \ell_n}{\partial \mathbf{w}}$. This computation is called *back-propagation* (of the derivatives). The gradient of $L_T(\mathbf{w})$ is the average (from equation (1)) of the gradients computed for each of the samples:

$$\frac{\partial L_T}{\partial \mathbf{w}} = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial \ell_n}{\partial \mathbf{w}} = \frac{1}{N} \sum_{n=1}^{N} \begin{bmatrix} \frac{\partial \ell_n}{\partial \mathbf{w}^{(1)}} \\ \vdots \\ \frac{\partial \ell_n}{\partial \mathbf{w}^{(K)}} \end{bmatrix}$$

(here, the derivative with respect to $\mathbf{w}$ is read as a column vector of derivatives). This average vector can be accumulated (see last assignment in Algorithm 1) as back-propagation progresses. For succinctness, operations are expressed as matrix-vector computations in Algorithm 1. In practice, the matrices would be very sparse, and correlations and explicit loops over appropriate indices are used instead.

Algorithm 1 Backpropagation

function $\nabla L_T \leftarrow \text{backprop}(T, \mathbf{w} = [\mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(K)}], \ell)$
  $\nabla L_T = \text{zeros}(\text{size}(\mathbf{w}))$
  for $n = 1, \ldots, N$ **do**
    $\mathbf{x}^{(0)} = \mathbf{x}_n$
    for $k = 1, \ldots, K$ **do** ▷ Forward propagation
      $\mathbf{x}^{(k)} \leftarrow f^{(k)}(\mathbf{x}^{(k-1)}, \mathbf{w}^{(k)})$ ▷ Compute and store layer outputs to be used in back-propagation
    end for
    $\nabla \ell_n = [\,]$ ▷ Initially empty contribution of the n-th sample to the loss gradient
    $\mathbf{g} = \frac{\partial \ell(y_n, \mathbf{x}^{(K)})}{\partial \mathbf{p}}$ ▷ $\mathbf{g}$ is $\frac{\partial \ell_n}{\partial \mathbf{x}^{(k)}}$
    for $k = K, \ldots, 2$ **do** ▷ Back-propagation
      $\nabla \ell_n \leftarrow [\mathbf{g} \frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{w}^{(k)}}; \nabla \ell_n]$ ▷ Derivatives are evaluated at $\mathbf{w}^{(k)}$ and $\mathbf{x}^{(k)}$
      $\mathbf{g} \leftarrow \mathbf{g} \frac{\partial \mathbf{x}^{(k)}}{\partial \mathbf{x}^{(k-1)}}$ ▷ Ditto
    end for
    $\nabla L_T \leftarrow \frac{(n-1)\nabla L_T + \nabla \ell_n}{n}$ ▷ Accumulate the average
  end for
end function

4 Stochastic Gradient Descent

In principle, a neural network can be trained by minimizing the training risk $L_T(\mathbf{w})$ defined in equation (1) by any of a vast variety of numerical optimization methods [5, 2]. At one end of the spectrum, methods that make no use of gradient information take too many steps to converge. At the other end, methods that use second-order derivatives (Hessian) to determine high-quality steps tend to be too expensive in terms of both space and time at each iteration, although some researchers advocate these types of methods [4]. By far the most widely used methods employ gradient information, computed by back-propagation [1]. Line search is too expensive, and the step size is therefore chosen according to some heuristic instead.

The *momentum method* [6, 8] starts from an initial value $\mathbf{w}_0$ chosen at random and iterates as follows:

$$\begin{aligned} \mathbf{v}_{t+1} &= \mu_t \mathbf{v}_t - \alpha \nabla L_T(\mathbf{w}_t) \\ \mathbf{w}_{t+1} &= \mathbf{w}_t + \mathbf{v}_{t+1} \,. \end{aligned}$$

The vector $\mathbf{v}_{t+1}$ is the *step* or *velocity* that is added to the old value $\mathbf{w}_t$ to compute the new value $\mathbf{w}_{t+1}$. The scalar $\alpha > 0$ is the *learning rate* that determines how fast to move in the direction opposite to the risk gradient $\nabla L_T(\mathbf{w})$, and the time-dependent scalar $\mu_t \in [0, 1]$ is the *momentum coefficient*. Gradient descent is obtained when $\mu_t = 0$. Greater values of μ_t encourage steps in a consistent direction (since the new velocity $\mathbf{v}_{t+1}$ has a greater component in the direction of the old velocity $\mathbf{v}_t$ than if no momentum were present), and these steps accelerate descent when the gradient of $L_T(\mathbf{w})$ is small, as is the case around shallow minima. The value of μ_t is often varied according to some schedule like the one in Figure 3. The rationale for the increasing values over time is that momentum is more useful in later stages, in which the gradient magnitude is very small as $\mathbf{w}_t$ approaches the minimum.

The learning rate α is often fixed, and is a parameter of critical importance [9]. A rate that is too large leads to large steps that often overshoot, and a rate that is too small leads to very slow progress. In practice, an initial value of α is chosen by cross-validation to be some value much smaller than 1. Convergence can

$$\mu_t = \min\left(\mu_{\max}, 1 - \frac{1}{2}\,\frac{1}{\lfloor \frac{t}{250} \rfloor + 1}\right)$$

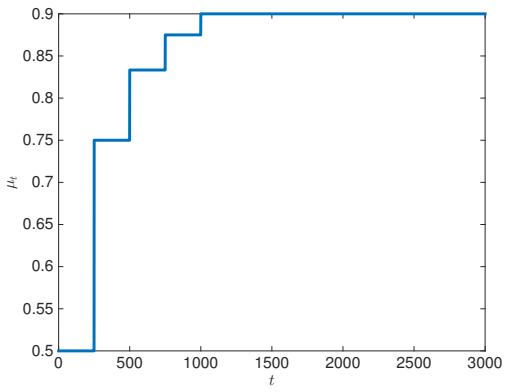

Figure 3: A possible schedule [8] for the momentum coefficient μ_t.

take between hours and weeks for typical applications, and the value of L_T is typically monitored through some user interface. When progress starts to saturate, the value of α is decreased (say, divided by 10).

Mini-Batches. The gradient of the risk $L_T(\mathbf{w})$ is expensive to compute, and one tends to use as large a learning rate as possible so as to minimize the number of steps taken. One way to prevent the resulting overshooting would be to do *online learning*, in which each step $\mu_t \mathbf{v}_t - \alpha \nabla \ell_n(\mathbf{w}_t)$ (there is one such step for each training sample) is taken right away, rather than accumulated into the step $\mu_t \mathbf{v}_t - \alpha \nabla L_T(\mathbf{w}_t)$ (no subscript n here). In contrast, using the latter step is called *batch learning*. Computing $\nabla \ell_n$ is much less expensive (by a factor of N) than computing ∇L_T. In addition—and most importantly for convergence behavior—online learning breaks a single batch step into N small steps, after each of which the value of the risk is re-evaluated. As a result, the online steps can follow very "curved" paths, whereas a single batch step can only move in a fixed direction in parameter space. Because of this greater flexibility, online learning converges faster than batch learning for the same overall computational effort. The small online steps, however, have high variance, because each of them is taken based on minimal amounts of data. One can improve convergence further by processing *mini-batches* of training data: Accumulate B gradients $\nabla \ell_n$ from the data in one mini-batch into a single gradient ∇L_T, take the step, and move on to the next mini-batch. It turns out that small values of B achieve the best compromise between reducing variance and keeping steps flexible. Values of B around a few dozen are common.

Termination. When used outside learning, gradient descent is typically stopped when steps make little progress, as measured by step size $\|\mathbf{w}_t - \mathbf{w}_{t-1}\|$ and/or decrease in function value $|L_T(\mathbf{w}_t) - L_T(\mathbf{w}_{t-1})|$. When training a deep network, on the other hand, descent is often stopped earlier to improve generalization. Specifically, one monitors the zero-one risk error of the classifier on a validation set, rather than the cross-entropy risk of the soft-max output on the training set, and stops when the validation-set error bottoms out, even if the training-set risk would continue to decrease. A different way to improve generalization, sometimes used in combination with early termination, is discussed in Section 5.

5 Dropout

Since deep nets have a large number of parameters, they would need impractically large training sets to avoid overfitting if no special measures are taken during training. Early termination, described at the end of the

previous section, is one such measure. In general, the best way to avoid overfitting in the presence of limited data would be to build one network for every possible setting of the parameters, compute the posterior probability of each setting given the training set, and then aggregate the nets into a single predictor that computes the average output weighted by the posterior probabilities. This approach, which is reminiscent of building a forest of trees, is obviously infeasible to implement for nontrivial nets.

One way to approximate this scheme in a computationally efficient way is called the *dropout* method [7]. Given a deep network to be trained, a *dropout network* is obtained by flipping a biased coin for each node of the original network and "dropping" that node if the flip turns out heads. Dropping the node means that all the weights and biases for that node are set to zero, so that the node becomes effectively inactive.

One then trains the network by using mini-batches of training data, and performs one iteration of training on each mini-batch after turning off neurons independently with probability $1 - p$. When training is done, all the weights in the network are multiplied by p, and this effectively averages the outputs of the nets with weights that depend on how often a unit participated in training. The value of p is typically set to $1/2$.

Each dropout network can be viewed as a different network, and the dropout method effectively samples a large number of nets efficiently.

Appendix: The Jacobians for Back-Propagation

If $f^{(k)}$ is a point function, that is, if it is $\mathbb{R} \to \mathbb{R}$, the individual entries of the Jacobian matrices (5) are easily found to be (reverting to matrix subscripts for the weights)

$$\frac{\partial x_i^{(k)}}{\partial W_{qj}^{(k)}} = \delta_{iq} \frac{df^{(k)}}{da_i^{(k)}} \tilde{x}_j^{(k-1)} \quad \text{and} \quad \frac{\partial x_i^{(k)}}{\partial x_j^{(k-1)}} = \frac{df^{(k)}}{da_i^{(k)}} W_{ij}^{(k)} \ .$$

The Kronecker delta

$$\delta_{iq} = \begin{cases} 1 & \text{if } i = q \\ 0 & \text{otherwise} \end{cases}$$

in the first of the two expressions above reflects the fact that $x_i^{(k)}$ depends only on the i-th activation, which is in turn the inner product of row i of $W^{(k)}$ with $\tilde{\mathbf{x}}^{(k-1)}$. Because of this, the derivative of $x_i^{(k)}$ with respect to entry $W_{qj}^{(k)}$ is zero if this entry is not in that row, that is, when $i \neq q$. The expression

$$\frac{df^{(k)}}{da_i^{(k)}} \quad \text{is shorthand for} \quad \left. \frac{df^{(k)}}{da} \right|_{a=a_i^{(k)}} ,$$

the derivative of the activation function $f^{(k)}$ with respect to its only argument a, evaluated for $a = a_i^{(k)}$.

For the ReLU activation function $h^k = h$,

$$\frac{df^{(k)}}{da} = \begin{cases} 1 & \text{for } a \geq 0 \\ 0 & \text{otherwise} \end{cases} \ .$$

For the ReLU activation function followed by max-pooling, $h^k(\cdot) = \pi(h(\cdot))$, on the other hand, the value of the output at index i is computed from a window $P(i)$ of activations, and only one of the activations (the one with the highest value) in the window is relevant to the output[2]. Let then

$$p_i^{(k)} = \max_{q \in P(i)} (h(a_q^{(k)}))$$

be the value resulting from max-pooling over the window $P(i)$ associated with output i of layer k. Furthermore, let

$$\hat{q} = \arg \max_{q \in P(i)} (h(a_q^{(k)}))$$

be the index of the activation where that maximum is achieved, where for brevity we leave the dependence of $\hat{q}$ on activation index i and layer k implicit. Then,

$$\frac{\partial x_i^{(k)}}{\partial W_{qj}^{(k)}} = \delta_{q\hat{q}} \frac{df^{(k)}}{da_{\hat{q}}^{(k)}} \tilde{x}_j^{(k-1)} \quad \text{and} \quad \frac{\partial x_i^{(k)}}{\partial x_j^{(k-1)}} = \frac{df^{(k)}}{da_{\hat{q}}^{(k)}} W_{\hat{q}j}^{(k)} \ .$$

[2] In case of a tie, we attribute the highest values in $P(i)$ to one of the highest inputs, say, chosen at random.

References

[1] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[2] S. Boyd and L. Vandeberghe. *Convex Optimization*. Cambdrige University Press, 2004.

[3] J. B. Hiriart-Urruty and C. Lemaréchal. *Convex analysis and minimization algorithms I: Fundamentals*, volume 305. Springer science & business media, 2013.

[4] J. Martens. Learning recurrent neural networks with Hessian-free optimization. In *Proceedings of the 28th International Conference on Machine Learning*, pages 735–742, 2011.

[5] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, NY, 1999.

[6] B. T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[7] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[8] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning*, pages 1139–1147, 2013.

[9] D. R. Wilson and T. R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16:1429–1451, 2003.