This document is a companion to a `C++` implementation of basic neural networks. We summarize here all the necessary formulas, and a quick derivation thereof.

Source: `github.com/dgdelmas`.

Background.

Gradient descent. Say we want to minimize some function $V(\boldsymbol{q})$. Then we can solve the ODE

$$\dot{\boldsymbol{q}} = -\nabla V \tag{1}$$

which is a flow equation whose critical points $\boldsymbol{q}_\star$ are those for which $\nabla V = 0$. Attractive critical points are minima, so the ODE will generically asymptote to them, and we are done. This is guaranteed to work if V is convex, but usually works even when it is not. (Of course, some V's will show runaway behavior or other pathologies; in practice we just cross our fingers).

If we apply the Euler method to the ODE above we end up with the usual iteration

$$\begin{aligned} \boldsymbol{q}_{n+1} &= \boldsymbol{q}_n + \alpha(-\nabla V(\boldsymbol{q}_n)) \\ &= \boldsymbol{q}_n - \alpha \boldsymbol{J}_n \end{aligned} \tag{2}$$

where α (defined by $\dot{\boldsymbol{q}} \approx \frac{\boldsymbol{q}_{n+1} - \boldsymbol{q}_n}{\alpha}$) is the learning rate and $J_i := \partial_i V$ is the (negative) force.

Note: it is sometimes a good idea to add a harmonic trap to $V(\boldsymbol{q})$, also known as *regularization*. Namely, if we want to minimize $V(\boldsymbol{q})$ but we don't want to allow $\boldsymbol{q}$ to become too large (say, to reduce overfitting), we can define[1]

$$\tilde{V}(\boldsymbol{q}) := V(\boldsymbol{q}) + \frac{1}{2}\lambda \boldsymbol{q}^2 \tag{3}$$

for a hyperparameter λ. We now minimize $\tilde{V}$ instead of V. For $\lambda = 0$ we find the old minimum $\tilde{\boldsymbol{q}}_\star = \boldsymbol{q}_\star$, while for $\lambda = \infty$ we find $\tilde{\boldsymbol{q}}_\star = 0$. For finite λ, we find an intermediate behavior, where $\tilde{\boldsymbol{q}}_\star$ is close to $\boldsymbol{q}_\star$ but the components are closer to the origin. (If we have reasons to believe that some finite $\boldsymbol{q}_0$ is preferred, then we use $(\boldsymbol{q} - \boldsymbol{q}_0)^2$ instead, of course). The iteration now looks like

$$\begin{aligned} \boldsymbol{q}_{n+1} &= \boldsymbol{q}_n - \alpha(\boldsymbol{J}_n + \lambda \boldsymbol{q}_n) \\ &= (1 - \alpha\lambda)\boldsymbol{q}_n - \alpha \boldsymbol{J}_n \end{aligned} \tag{4}$$

So in order to regularize gradient descent, we just replace the coefficient of $\boldsymbol{q}$ in the previous iteration by a slightly smaller number.

If $\boldsymbol{q}$ lives in a very high dimensional space, we need not worry about local minima[2] but saddle points significantly slow down convergence. If we ever get close to one, J_i will be small and the evolution stays close to this point for many iterations. A way to overcome this is to give the particle *inertia*. This will allow $q_i(t)$ to keep moving even if J_i is small.[3] We achieve this by replacing the ODE by

$$\ddot{\boldsymbol{q}} = -\boldsymbol{J} - \gamma \dot{\boldsymbol{q}} \tag{5}$$

which now has two degrees of freedom per variable, $q_i, \dot{q}_i$. Now, even if J_i is small, $\ddot{q}_i$ will be finite as long as $\dot{q}_i$ is, namely even if there is no force, the particle will keep moving if it already had some velocity.

[1] This is known as L_2-reg, for obvious reasons, but we can of course replace $|\cdot|_2$ by any other norm.

[2] An intuitive argument is as follows. Assume that $\boldsymbol{q}_\star$ is a critical point, and write $\boldsymbol{q} = \boldsymbol{q}_\star + \boldsymbol{\delta}$. Then, the critical point is a local minimum if V grows with all δ_i, a local maximum if it decreases with every δ_i, and a saddle point if it grows with some and decreases with others. If V is "generic", then the probability that all $\partial_i V$ have the same sign is $2^{-d} \ll 1$. So most critical points are saddle points.

[3] This also has the advantage that the iteration will jump over thin, steep minima, which are often considered to be overfitted (since they tend to be very unstable and mostly just artifacts of specific sets of data rather than a generic property of the underlying system).

Importantly, the friction term $-\gamma\dot{q}_i$ prevents us from oscillating around a minimum, it forces the particle to "fall down" as it loses energy. Again, the particle asymptotes to a local minimum, since $\boldsymbol{q}$ stops evolving only when $\dot{\boldsymbol{q}} = \ddot{\boldsymbol{q}} = 0$.

We can write the ODE in first order form as

$$\begin{aligned} \dot{\boldsymbol{v}} &= -\boldsymbol{J} - \gamma \boldsymbol{v} \\ \dot{\boldsymbol{q}} &= \boldsymbol{v} \end{aligned} \tag{6}$$

which, applying Euler once again, yields the "momentum" iteration

$$\begin{aligned} \boldsymbol{v}_{n+1} &= -\alpha \boldsymbol{J}_n + (1 - \gamma\alpha)\boldsymbol{v}_n \\ \boldsymbol{q}_{n+1} &= \boldsymbol{q}_n + \alpha \boldsymbol{v}_{n+1} \end{aligned} \tag{7}$$

which is the usual update rule (under the relabelling $\boldsymbol{m} := -\gamma\boldsymbol{v}$, $\beta := 1 - \gamma\alpha$). This requires more memory than naive gradient descent, but it is more robust.

Improvements. Let us forget about momentum for a minute, and improve the original update rule by keeping track of the size of J_i. We will then combine with momentum to get Adam.

The idea is that J_i might be large for some i and small for some other, in which case the latter parameters will learn very slowly. But we cannot simply increase the learning rate because then, the former would explode. The solution is to have an estimate of the order of magnitude of J_i for each i, say $a_i = E_t[J_i(\boldsymbol{q}_t)] := \frac{1}{t}\int_{t' \leq t} J_i(\boldsymbol{q}_{t'})$, and use local effective rates $\alpha_i := \alpha/a_i$. This way, if J_i is large, α_i is small, and vice versa, and all parameters learn more or less at the same rate.

This naive version does not work because J_i is not sign definite, so there are cancellations and $E_t[J_i]$ is not a good estimate of the order of magnitude of J_i. So we use instead $E_t[J_i^2]^{1/2}$. This leads to the modified update rule known as AdaGrad, which replaces naive

$$q_{i,n+1} = q_{i,n} - \alpha J_{i,n} \tag{8}$$

by

$$\begin{aligned} q_{i,n} &= q_{i,n-1} - \frac{\alpha}{a_{i,n}^{1/2}} J_{i,n} \\ a_{i,n} &= \frac{1}{n}\sum_{m=1}^{n} J_{i,m}^2 \\ &= \frac{n-1}{n} a_{i,n-1} + \frac{1}{n} J_{i,n}^2 \end{aligned} \tag{9}$$

This update rule is just naive Euler, but using the effective learning rate $\alpha/a_i^{1/2}$.

Note: in practice, it is better to use $\alpha/(a_{i,n}^{1/2} + \epsilon)$ instead of $\alpha/a_{i,n}^{1/2}$, where $\epsilon \sim 10^{-8}$ is a small shift to avoid division by zero if $a_{i,n}$ happens to be too small.

We can actually do slightly better: instead of using a regular moving average, we use an exponentially weighted one,[4] to give more weight to current values of J_i. This is called RMSProp, and it reads

$$\begin{aligned} q_{i,n+1} &= q_{i,n} - \frac{\alpha}{e_{i,n}^{1/2}} J_{i,n} \\ e_{i,n} &= \beta e_{i,n-1} + (1-\beta) J_{i,n}^2 \end{aligned} \tag{10}$$

[4]Given a sequence x_n, the moving average is defined by $a_n := \frac{1}{n}(x_1 + \cdots + x_n)$ or, equivalently, by $a_n = \frac{1}{n}x_n + \frac{n-1}{n}a_{n-1}$. All data points up to n are given equal weight. Alternatively, we could give larger weight to later points: $e_n := (1-\beta)(\beta^{n-1}x_1 + \beta^{n-2}x_2 + \cdots + \beta x_{n-1} + x_n)$ where $\beta \in [0,1]$ is some fixed parameter. Equivalently, $e_n = (1-\beta)x_n + \beta e_{n-1}$: the exponentially weighted moving average e_n is the weighted average between the current value x_n, and the previous value e_{n-1}. Much like a_n, e_n gives a smooth version of x_n, but "old" values x_k for $k \ll n$ are given exponentially small weight $e_n = (1-\beta)\sum_{k=1}^{n}\beta^{n-k}x_k$.

The "decay rate" β is usually set to 0.9, so e_i is very smooth but it lags J_i^2 significantly.

Note: a slightly more correct version, but also much more computationally intensive, is to use the matrix $\boldsymbol{J}_n\boldsymbol{J}_n^T$ instead of just the diagonal components $J_{i,n}^2$. In practice it does not seem to be worth the extra complexity.

We can finally combine this adaptive learning rate with the inertia added by the momentum method to get the best of both worlds. This leads to the Adam method. It reads

$$\begin{aligned} m_{i,n} &= \beta_1 m_{i,n-1} + (1-\beta_1) J_{i,n} \\ v_{i,n} &= \beta_2 v_{i,n-1} + (1-\beta_2) J_{i,n}^2 \\ q_{i,n} &= q_{i,n-1} - \frac{\alpha}{\sqrt{\frac{v_{i,n}}{1-\beta_2^n}}} \frac{m_{i,n}}{1-\beta_1^n} \end{aligned} \tag{11}$$

which is like the momentum iteration but we use adaptive learning rates $\sim \alpha/v_i^{1/2}$ (and we also use unbiased estimates, due to the division by $1-\beta^n$).

A different direction in which one could try to improve gradient descent is to use a higher order solver (e.g. Runge-Kutta), by keeping track of higher derivatives. For example, we could replace naive Euler

$$\boldsymbol{q}_{n+1} = \boldsymbol{q}_n - \alpha \boldsymbol{J}_n \tag{12}$$

by

$$\boldsymbol{q}_{n+1} = \boldsymbol{q}_{n-1} - 2\alpha \boldsymbol{J}_n \tag{13}$$

or

$$\boldsymbol{q}_{n+1} = -\tfrac{3}{2}\boldsymbol{q}_n + 3\boldsymbol{q}_{n-1} - \tfrac{1}{2}\boldsymbol{q}_{n-2} - 3\alpha \boldsymbol{J}_n \tag{14}$$

This requires more memory but might allow for larger learning rates, leading to fewer iterations. I haven't tested if this works better, nor have I looked at the literature to check if it has already been discussed, although it most likely has.

Logistic regression. Say we want to classify samples, and it is not feasible or useful to write procedural code. So we are given instead labelled data $\{(x_s, y_s)\}_{s\in S}$, which we will use to fit some model of our choosing and, if the data is a statistically good representation of the underlying system, and our model a good representation of its dynamics, we can then predict a class for new, unseen data.

A simple example is logistic regression. Say there are only two classes (which doesn't lose us much generality: for more classes we can use various two-class representations, such as binary or one-hot). We label these as $y=0$ and $y=1$. Then, our model is taken to be $p(x) := \sigma(w\cdot x + b)$, where w is a vector of "weights" and b is a real number, the "bias". Furthermore, $\sigma : \mathbb{R} \to [0,1]$ is some of "activation", such as a logistic $\sigma(z) = 1/(1+e^{-z})$ or a soft version thereof, $\sigma(z) = (|x| + x + 1)/(2|x| + 2)$.

We use the dataset S to estimate w, b, such that $p(x_s)$ is as close as possible to y_s for all $s \in S$, according to some metric. For example, we can define $L(w,b) := \sum_{s\in S} |p(x_s) - y_s|^2$ and minimize L using gradient descent. This way we find the optimal weights and biases $w_\star, b_\star$ and we can use $p_\star(x)$ to predict the class of new examples.

Note that $p(x) \in [0,1]$ is a probability, so $p(x)$ is an estimate of how confident we are that $y=1$. For example, we can use $\sigma = 1/2$ to be the division boundary, such that $p < 1/2$ is declared to correspond to the prediction $y=0$, and $p > 1/2$ to $y=1$. So we separate the data linearly, via $wx + b = \sigma^{(-1)}(1/2)$.

The closer β is to 1, the smoother e_n is, but also the more it lags x_n. The closer β is to 0, the closer e_n is to x_n, and therefore the rougher it becomes.

Two things to note. First, in the regular moving average the weights add up to 1: $\frac{1}{n}\sum_{k=1}^n = 1$. This is not the case for the exponentially weighted average: $(1-\beta)\sum_{k=1}^n \beta^{n-k} = 1-\beta^n$. Second, as defined above, the first few terms are $e_1 = (1-\beta)x_1$, $e_2 = \beta(1-\beta)x_1 + (1-\beta)x_2$, etc., which, unless $1-\beta \approx 1$, are significantly smaller than x_1, x_2, etc. Hence, this moving average largely underestimates x_n for small n, especially for β close to 1 (which is the case in practice).

We can solve both issues by using the "unbiased" estimate $\hat{e}_n := e_n/(1-\beta^n)$.

What happens if the data doesn't happen to be linearly separable? We can still use logistic regression if we use feature maps, namely we "pre-process" the data, $x \to \phi(x)$ for some choice of non-linear ϕ. For example, $\phi(x_1, x_2) = (x_1\ x_2\ x_1^2\ x_1x_2\ x_2^2)$. So our model is instead

$$p(x) = \sigma(w \cdot \phi(x) + b) \tag{15}$$

and now the decision boundary is the non-linear $w\phi(x) + b = \sigma^{(-1)}(1/2)$. The tricky part is to choose the features without overfitting (e.g., monomials can fit any function but they are very volatile, while e.g. trigonometric functions stay bounded but can only fit periodic data, so they have very high bias).

Wouldn't it be nice if our model could learn ϕ instead of having to choose it by hand? This is kind of what neural networks do for us: instead of feeding x to p, we feed a separate logistic, like so:

$$p(x) = \sigma(w \cdot \phi(x) + b), \qquad \phi(x) = \tilde{\sigma}(\tilde{w} \cdot x + \tilde{b}) \tag{16}$$

for some choice of $\tilde{\sigma}$, which could be equal to σ or some other function $\mathbb{R} \to \mathbb{R}$. We now fit both w, b and $\tilde{w}, \tilde{b}$. This is roughly equivalent to regular regression, but with a feature map that is itself another regression. This allows for non-linear decision boundaries without the need to manually choose feature maps.

Of course, we don't have to stop here: we can feed yet another logistic to the model, so that the feature map itself contains another learnable feature map. This way we get a network of logistic regressions, where each layer functions as a feature map for the following one. This allows us to have models with arbitrarily large number of adjustable parameters, without increasing the variance too much – hopefully at least. This is called a *neural network*.

Neural networks.

Dense networks. Consider a fully connected network. Denote by $p_\alpha^{(\ell)}$ the output of the α-th neuron in the ℓ-th layer, where $\ell = 0, 1, \ldots, L-1$ (with L the depth of the network) and $\alpha = 0, 1, \ldots, n_{\text{out}}^{(\ell)} - 1$. Concretely,

$$p_\alpha^{(\ell)} = \sigma^{(\ell)}(w_\alpha^{(\ell)} \cdot p^{(\ell-1)} + b_\alpha^{(\ell)}) \tag{17}$$

where

$$w_\alpha^{(\ell)} \cdot p^{(\ell-1)} = \sum_{\beta=0}^{n_{\text{in}}^{(\ell)}-1} w_{\alpha\beta}^{(\ell)} p_\beta^{(\ell-1)} \tag{18}$$

where of course $n_{\text{in}}^{(\ell)} = n_{\text{out}}^{(\ell-1)}$. Also $\sigma^{(\ell)} \colon \mathbb{R} \to \mathbb{R}$ is some choice of activation, e.g. a logistic function.

Note: strictly speaking, (17) doesn't make sense for the first layer $\ell = 0$, since $p^{(\ell-1)}$ is then undefined. Of course, the first layer takes an input vector x, and by $p^{(-1)}$ we mean this vector.

The input size of the network is defined as the input size of its first layer, $n_{\text{in}} := n_{\text{in}}^{(0)}$, and its output size is defined as the output size of its last layer, $n_{\text{out}} := n_{\text{out}}^{(L-1)}$.

The output of the network for a given input x is $p(x) := p^{(L-1)}(x)$ and we want to choose $\{w, b\}$ so as to minimize the empirical cost estimate

$$L = \frac{1}{|S|} \sum_{s \in S} L_s, \qquad L_s := \ell(\zeta(p(x_s)), y_s) \tag{19}$$

where $\ell \colon \mathbb{R}^{n_{\text{out}}} \times \mathbb{R}^{n_{\text{out}}} \to \mathbb{R}$ is some choice of cost function, and S denotes the sample space. Here ζ denotes a "post-processing" function $\zeta \colon \mathbb{R}^{n_{\text{out}}} \to \mathbb{R}^{n_{\text{out}}}$, e.g. soft-max:

$$\zeta_\alpha(z) := \frac{e^{z_\alpha}}{\sum_\beta e^{z_\beta}} \tag{20}$$

A simple choice for the cost function is $\ell(v_1, v_2) = \frac{1}{2}(v_1 - v_2)^2$, while for binary output $y \in \{0, 1\}$ another common choice is cross-entropy[5]

$$\ell(v_1, v_2) = -v_2 \cdot \log(v_1) - (1 - v_2) \cdot \log(1 - v_1) \tag{21}$$

(where everything is done entry-wise.)

For simplicity we will assume that all the outputs have the same character (same type and same order of magnitude) and the overall cost weights them all evenly and with the same local loss:

$$\ell(\zeta(p(x_s)), y_s) = \frac{1}{n_{\text{out}}} \sum_{\alpha=0}^{n_{\text{out}}-1} \tilde{\ell}(\zeta_\alpha(p(x_s)), y_{s\alpha}) \tag{22}$$

where $\tilde{\ell}\colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. If the various components of the output have different characters (such as the location of an animal in a picture *and* a label for that animal), then we have to choose how much weight we give to the different components, and this is yet another hyper-parameter (which is actually very hard to choose, since – unlike other hyper-parameters – it changes the value of the cost function and it's therefore hard to compare performance for different choices).

We now choose our weights and biases such that L is minimized; we do this using gradient descent. In order to implement gradient descent we need the derivatives $L_{,\theta} := \partial L/\partial\theta$, where $\theta \in \{w^{(\ell)}_{\alpha\beta}, b^{(\ell)}_\alpha\}$. We write

$$L_{,\theta} = \frac{1}{|S|} \sum_{s \in S} L_{s,\theta}, \qquad L_{s,\theta} := \frac{\partial L_s}{\partial \theta} \tag{23}$$

Clearly, given that L depends on w, b only via the combination $w \cdot p^{(\ell-1)} + b$, we have

$$(L_{s,w})^{(\ell)}_{\beta\gamma} = (L_{s,b})^{(\ell)}_\beta p^{(\ell-1)}_\gamma \tag{24}$$

and therefore it suffices to compute $(L_{s,b})^{(\ell)}_\beta$. By definition,

$$\begin{aligned}(L_{s,b})^{(\ell)}_\beta &= \frac{1}{n_{\text{out}}} \frac{\partial}{\partial b^{(\ell)}_\beta} \sum_{\alpha=0}^{n_{\text{out}}-1} \tilde{\ell}(\zeta_\alpha(p(x_s)), y_{s\alpha}) \\ &= \frac{1}{n_{\text{out}}} \sum_{\alpha,\tau=0}^{n_{\text{out}}-1} \tilde{\ell}'(\zeta_\alpha, y_\alpha) \frac{\partial \zeta_\alpha}{\partial p_\tau} \frac{\partial p_\tau}{\partial b^{(\ell)}_\beta}\end{aligned} \tag{25}$$

Thus, we need to find the derivative $\partial p_\tau / \partial b^{(\ell)}_\beta$. If we differentiate (17) we find

$$\frac{\partial p^{(\ell)}_\alpha}{\partial b^{(\ell')}_\beta} = q^{(\ell)}_\alpha \frac{\partial}{\partial b^{(\ell')}_\beta} [w^{(\ell)}_\alpha \cdot p^{(\ell-1)} + b^{(\ell)}_\alpha] \tag{26}$$

where

$$q^{(\ell)}_\alpha := \sigma^{(\ell)\prime}(w^{(\ell)}_\alpha \cdot p^{(\ell-1)} + b^{(\ell)}_\alpha) \tag{27}$$

The derivative above satisfies a simple recurrence relation:

$$\frac{\partial}{\partial b^{(\ell')}_\beta} [w^{(\ell)}_\alpha \cdot p^{(\ell-1)} + b^{(\ell)}_\alpha] = \begin{cases} \delta_{\alpha\beta} & \ell' = \ell \\ w^{(\ell)}_{\alpha\gamma} \dfrac{\partial p^{(\ell-1)}_\gamma}{\partial b^{(\ell')}_\beta} & \ell' < \ell \end{cases} \tag{28}$$

[5]It is often the case that the output describes a probability function, namely $\sum_\alpha y_\alpha = 1$ (such as in a one-hot representation

(with an implicit sum over γ).

This means that the function $L_{s,b}$ satisfies its own recurrence relation:

$$(L_{s,b})_\beta^{(\ell)} = q_\beta^{(\ell)} \sum_{\gamma=0}^{n_{\text{out}}^{(\ell+1)}-1} w_{\gamma\beta}^{(\ell+1)} (L_{s,b})_\gamma^{(\ell+1)} \tag{29}$$

with initial condition

$$(L_{s,b})_\beta^{(L-1)} = \frac{1}{n_{\text{out}}} \sum_{\alpha=0}^{n_{\text{out}}-1} \tilde{\ell}'(\zeta_\alpha, y_\alpha) \frac{\partial \zeta_\alpha}{\partial p_\beta} q_\beta^{(\ell)} \tag{30}$$

and we are done. We use (30) to initialize $L_{s,b}$, back-propagate using (29), and use the chain rule (24) to find $L_{s,w}$. We now have all the derivatives of the cost function, and we can update w, b using gradient descent. After a few iterations, we will have found the optimal $w_\star, b_\star$ for our network, and we can use it to predict new examples.

Note: in practice, ζ is either the identity or soft-max. In the first case, $\partial \zeta_\alpha / \partial p_\beta = \delta_{\alpha\beta}$, while in the second case

$$\frac{\partial \zeta_\alpha}{\partial v_\beta} = \tfrac{1}{D}\big(\delta_{\alpha\beta} e^{v_\alpha} - \tfrac{1}{D} e^{v_\alpha + v_\beta}\big) \equiv \zeta_\alpha(\delta_{\alpha\beta} - \zeta_\beta) \tag{31}$$

where $D := \sum_\beta e^{v_\beta}$; this is a rank-1 update to a diagonal matrix, so it is a cheap operation. In any case, without soft-max, (30) reads

$$(L_{s,b})_\beta^{(L-1)} = \frac{1}{n_{\text{out}}} \tilde{\ell}'(p_\beta, y_\beta) q_\beta^{(\ell)} \tag{32}$$

while with soft-max, it becomes

$$(L_{s,b})_\beta^{(L-1)} = \frac{\zeta_\beta}{n_{\text{out}}} \left(\tilde{\ell}'(\zeta_\beta, y_\beta) - \sum_{\alpha=0}^{n_{\text{out}}-1} \tilde{\ell}'(\zeta_\alpha, y_\alpha) \zeta_\alpha \right) q_\beta^{(\ell)} \tag{33}$$

which barely increases the time complexity.

Convolutional networks. A conv net is a special case of a dense network, in which we manually impose some weights to vanish, and some others to be identical to each other, drastically reducing the number of trainable parameters. This leads to much higher bias – it performs much better than a regular network if these constraints are roughly satisfied by the underlying system to model, and much worse if not. The former is the case e.g. for image recognition: a regular network would treat all pixels democratically, while a conv net would treat nearby pixels differently from far away ones.

The following could be an intuitive way to understand how a conv net adds a prior to a dense net. Consider a rank-3 array A_{ijk} of dimension $m \times n \times p$. We can flatten this to a one-dimensional array A_I, where e.g. $I = npi + pj + k$. Conversely, we can unroll a one-dimensional array A_i into a multidimensional one $A_{i_1 i_2 \dots i_k}$ according to some choice of bijection $\mathbb{N} \to \mathbb{N}^k$. This is just a different representation of the same data, one that might be useful in some special cases. With this in mind, take a dense layer $p_i = \sigma(\omega_{ij} x_j + b_i)$ and unroll the index i into, say, a pair of indices ab:

$$p_{ab} = \sigma(\omega_{abcd} x_{cd} + b_{ab}) \tag{34}$$

(with implicit sum over repeated indices.) This is still the same layer, just in a slightly awkward notation. Next, we impose *by hand* that ω, b have a certain structure, e.g. $\omega_{abcd} = \Omega_{ac}\delta_{bd}$, $b_{ab} = B_a$. This greatly increases the bias and reduces the number of parameters the network has to learn. It increases efficiency *if* our choice of map $i \to \{a, b\}$ *and* our choice of constrains on ω, b happen to reasonably capture some property

of categorical data). Moreover, if we apply soft-max, then so does our prediction $p(x)$. Then, cross entropy simply measures the distance between these two probability distributions.

of the underlying data x_i. This is precisely what a conv net does: it unrolls the data into higher dimensional arrays, and then imposes on the weights and biases a specific choice of structure, reducing their rank (and with it the number of degrees of freedom). We elaborate on this next.

By definition, the input to a conv layer is a certain rank-3 array x_{ijk}, and the output is another rank-3 array p_{ijk}:

$$p_{ijk} = \sigma\left(\sum_{abc} f_{iabc} x_{a,b+js_v,c+ks_h} + b_i\right) \tag{35}$$

where s_v, s_h are a pair of integers that define the layer – these are called the vertical and horizontal stride, respectively. Moreover, we use f to denote the weights since we will call this layer a "filter". As we can see, this is a special case of a dense layer, where the weights and biases are constrained to take a very specific form. In the dense-layer notation,

$$p_{ijk} = \sigma(w_{ijk,abc} x_{abc} + \tilde{b}_{ijk}) \tag{36}$$

where[6]

$$\begin{aligned} w_{ijk,abc} &:= \sum_{b'c'} f_{iab'c'} \delta_{b,b'+js_v} \delta_{c,c'+ks_h} \\ &= f_{ia,b-js_v,c-ks_h} \\ \tilde{b}_{ijk} &:= b_i \end{aligned} \tag{37}$$

Therefore, a conv net is the same as a regular layer, except that instead of all-to-all weights $w_{ijk,abc}$ we have a much smaller set of weights $f_{ia,b-js_v,c-ks_h}$, and instead of one bias per input $\tilde{b}_{ijk}$, we have one bias per *channel* b_i.

Let us be more specific with dimensions of arrays. The input x is taken to be of dimension i_d, i_v, i_h, where d, v, h stand for "depth", "vertical", and "horizontal", respectively, and i stands for "input". We imagine x as a $3d$ matrix of height i_v, width i_h, and depth i_d.

The input size $i_d \times i_v \times i_h$ is determined by the data we have. In the case of dense layers, we get to choose the number of neurons; in the case of a conv net, we get to choose the number of filters and their size and stride. We denote the number of filters by o_d, and their size by $f_v \times f_h$. The size of the output is fixed in terms of these parameters; we write the explicit formula in the next paragraph.

A conv layer will consist of a sequence of filters f_i, where each filter is also a rank-3 array f_{iabc}. i can take o_d values; a can take i_d values; and b, c can take f_v, f_h values, respectively. In other words, f is a rank-4 array of dimension $o_d \times i_d \times f_v \times f_h$. The explicit, full form of the activation operation is

$$p_{ijk} = \sigma\left(\sum_{a=0}^{i_d-1} \sum_{b=0}^{f_v-1} \sum_{c=0}^{f_h-1} f_{iabc} x_{a,b+js_v,c+ks_h} + b_i\right), \quad \begin{cases} i = 0, 1, \ldots, o_d - 1 \\ j = 0, 1, \ldots, o_v - 1 \\ k = 0, 1, \ldots, o_h - 1 \end{cases} \tag{38}$$

where o_v satisfies $i_v - 1 = (f_v - 1) + (o_v - 1)s_v$, i.e., $o_v = (i_v - f_v)/s_v + 1$, and similarly for o_h. If this is not a integer, we simply declare that the filter is not compatible with the input. We could also pad, but we won't do that here.

Note: in the expressions above, o stands for "output", since the output p_{ijk} is a rank-3 array of dimension $o_d \times o_v \times o_h$.

A network simply takes p_{ijk} as the input to a second conv layer, whose output is the input to a third layer, etc. Usually, we flatten the end result, and feed it to a small-ish dense network to do the final classification for us. The input to the very first layer consists of a rank-3 array x_{ijk} where the first index can take three values, for the three color channels, and jk specify a pixel in a certain image. For hidden layers the different channels are not particularly interpretable. In any case, each face of p represents a different channel of the same picture.

[6]When dealing with convolutions, it is notationally convenient to extends all sums over the whole integers, $\sum_i v_i = \sum_{i=-\infty}^{\infty} v_i$, and simply declare that $v_i = 0$ when i is out of range. So by $f_{ia,b-js_v,c-ks_h}$ we actually mean $f_{ia,b-js_v,c-ks_h} \times [0 \le b - js_v < f_v] \times [0 \le c - ks_h < f_h]$, with f_v, f_h to be defined below.

A sequence of convolutional layers looks like so:

$$p^{(\ell)}_{ijk} = \sigma^{(\ell)}\left(\sum_{abc} f^{(\ell)}_{iabc} p^{(\ell-1)}_{a,b+js^{(\ell)}_v,c+ks^{(\ell)}_h} + b^{(\ell)}_i\right) \tag{39}$$

where as usual $p^{(-1)}$ denotes the input to the network. Naturally, $i^{(\ell)}_d = o^{(\ell-1)}_d$, and similarly for i_v, i_h. From now on, we drop the (ℓ) superscript in the stride parameters to lighten the notation.

Finally, we connect a series of conv layers with a series of dense layers

$$x \to C^{(0)} \to C^{(1)} \to \cdots \to C^{(n_c-1)} \to D^{(n_c)} \to D^{(n_c+1)} \to \cdots \to D^{(L-1)} \equiv p \tag{40}$$

where C stands for convolutional and D for dense. The only tricky part is to connect $C^{(n_c-1)}$ to $D^{(n_c)}$, which we do via flattening the rank-3 array $p^{(n_c-1)}_{ijk}$ to a linear array x_I with the usual bijection

$$I = o_v o_h i + o_h j + k \qquad \Leftrightarrow \qquad \begin{cases} k = I \mod o_h \\ j = (I-k)/o_h \mod o_v \\ i = (I-k-o_h j)/(o_h o_v) \end{cases} \tag{41}$$

where o_v, o_h are the vertical and horizontal sizes of the output to the last conv layer $C^{(n_c-1)}$. Specifically, we define

$$x_I = p^{(n_c-1)}_{i(I),j(I),k(I)} \tag{42}$$

and feed x_I as a linear input to the dense layer $D^{(n_c)}$.

We define a loss function as usual, and compute the various derivatives $L_{,\theta}$, where now $\theta \in \{b_\alpha, w_{\alpha\beta}, f_{aijk}\}$. The old formulas for the dense layers $D^{(n_c)}, \ldots, D^{(L-1)}$ are still valid, and we now need the analogues for the conv layers $C^{(0)}, \ldots, C^{(n_c-1)}$. Again, given that the output depends on f, b only via the combination $f \cdot p + b$, it is enough to compute the derivative with respect to the biases. We do this next.

First, we note that backprop reads

$$\frac{\partial p^{(\ell)}_{ijk}}{\partial b^{(\ell')}_\alpha} = \begin{cases} q^{(\ell)}_{ijk}\delta_{i\alpha} & \ell = \ell' \\ q^{(\ell)}_{ijk} \sum_{abc} f^{(\ell)}_{iabc} \dfrac{\partial p^{(\ell-1)}_{a,b+js_v,c+ks_h}}{\partial b^{(\ell')}_\alpha} & \ell' < \ell \end{cases} \tag{43}$$

where

$$q^{(\ell)}_{ijk} := \sigma^{(\ell)\prime}\left(\sum_{abc} f^{(\ell)}_{iabc} p^{(\ell-1)}_{a,b+js_v,c+ks_h} + b^{(\ell)}_i\right) \tag{44}$$

Therefore, if we define

$$\Xi^{(\ell)}_{ijk} = \sum_{a'b'c'} q^{(\ell)}_{ijk} f^{(\ell+1)}_{a',i,j-b's_v,k-c's_h} \Xi^{(\ell+1)}_{a'b'c'} \tag{45}$$

with initial condition

$$\Xi^{(n_c-1)}_{ijk} = \sum_\beta q^{(n_c-1)}_{ijk} (L_{s,b})^{(n_c)}_\beta w^{(n_c)}_{\beta,I(ijk)} \tag{46}$$

then the derivatives are given by

$$\begin{aligned} (L_{s,b})^{(\ell)}_i &= \sum_{jk} \Xi^{(\ell)}_{ijk} \\ (L_{s,f})^{(\ell)}_{iabc} &= \sum_{jk} \Xi^{(\ell)}_{ijk} p^{(\ell-1)}_{a,b+js_v,c+ks_h} \end{aligned} \tag{47}$$

and we are done.