3 Supplemental Training Procedures

Objective

The previous chapter focused on deep multilayer network training as an optimization problem, in which the weights and biases of the network are chosen to optimize the performance function. This chapter looks at procedures that can be used to supplement the basic optimization algorithms. These supplementary procedures have two purposes. The first purpose is to select a good starting point for the optimization process, which will speed up the convergence of the optimization algorithm. The second purpose is to ensure that the final trained network will perform as well on new data as it did on the training data.

Theory and Examples

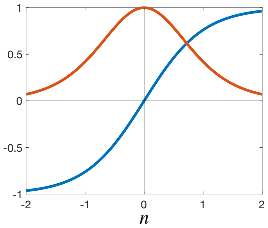
This chapter discusses techniques that can be used to supplement the training algorithms that were discussed in the last chapter. These techniques can: 1) speed up the convergence of the training algorithms, and 2) produce networks that will generalize well to new situations.

Some of the material in this chapter is covered in Chapters 13 and 22 of NND2. Here we review the basic concepts and discuss specific topics that are especially important for deep networks.

Speeding Up Convergence

As we mentioned in previous chapters, much of the basic theory for deep networks was developed many years ago. For example, the backpropagation algorithm for computing the gradient of network performance with respect to the network weights and biases was first popularized in the mid 1980s. However, the extensive use of neural networks with many layers did not occur until training times became manageable. Two developments, in particular, improved training times: the introduction of the cuda language for general purpose programming of GPUs, and the discovery of methods to overcome the vanishing gradient problem.

The *vanishing gradient* problem occurs when a sigmoid (S-shaped) activation function is used in a network with a large number of layers (or in recurrent networks). The figure in the margin shows the hyperbolic tangent sigmoid function (blue) and its derivative (red). As the net input n becomes large in magnitude, the derivative decreases quickly in size. (The net input will become large if the input to the layer, or the layer weights, become large.) When the sigmoid saturates, the derivative will be very small. Since we multiply by the derivative in the backpropagation process to compute the gradient, the gradient will also become small.



The sigmoid *squashes* the input into the range $[-1, 1]$ and must become flat for large net inputs. When multiple sigmoid layers are cascaded, these multiple squashing operations cause more flat areas in the overall network response. When the gradient is computed by backpropagating through the layers, its magnitude will generally become smaller and smaller, slowing the training convergence.

The flattening operation produced by cascading sigmoid layers is illustrated in Solved Problem P6.3, which demonstrates what happens as the number of sigmoid layers is increased. The process is also shown in Figure 3.1. With a two layer network (with a sigmoid hidden layer), the network response gradually decays to zero. As the number of layers increases, the network response becomes almost flat (derivative almost zero) over a larger portion of the relevant function domain.

To experiment with increasing the number of layers in a network, use the Deep Learning Demonstration Cascaded Function (**dl2cf**).

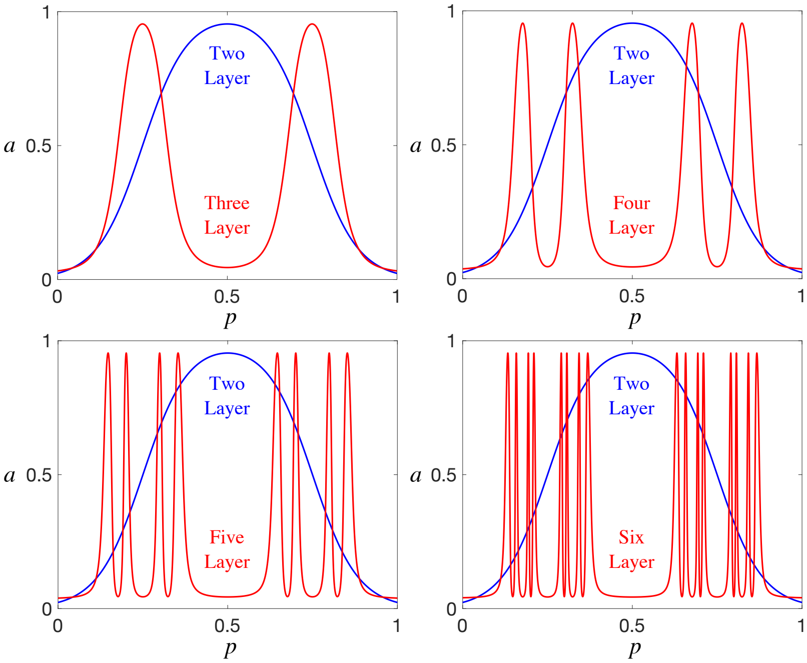


Figure 3.1: Effect of Cascading Sigmoid Layers

Much of the work to make deep network training practical has focused on mitigating the vanishing gradient problem. One very helpful innovation was the use of the *ReLU* (or *poslin*) activation function, which was discussed in Chapter 6. Figure 3.2 demonstrates the operation of the same network used for Figure 3.1, but with the *tansig* layers replaced with *poslin* layers. We no longer have the large areas with diminished slope.

In addition to the introduction of the *ReLU* activation function, much additional research has focused on overcoming the vanishing gradient problem. This additional work emphasized two interrelated approaches: normalization and weight initialization.

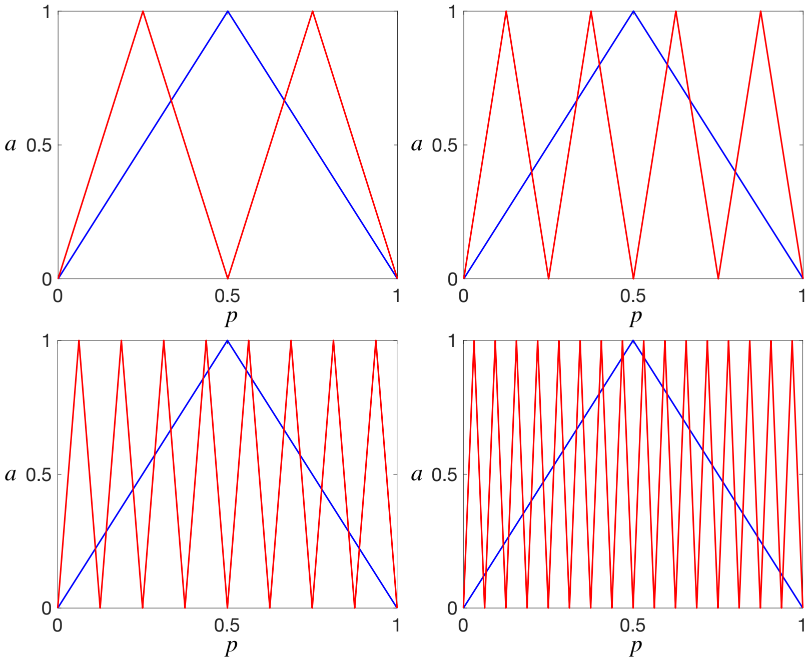


Figure 3.2: Effect of Cascading ReLU (poslin) Layers

Normalization involves pre-processing the input to a layer so that it has a standard distribution. If the inputs to a layer are always of the same size, it is easier to adjust the initial weights so that the net input to the activation function will be in a reasonable range.

Normalization was first introduced for the network inputs. As described in Chapter 22 of NND2, there are two common types of normalization. The first type normalizes the input so that all values fall in some set range – typically $[-1, 1]$. This can be done using

$$\check{\mathbf{p}} = \left(\mathbf{p} - \mathbf{p}^{min}\right) \oslash \left(\mathbf{p}^{max} - \mathbf{p}^{min}\right) - 1 \tag{3.1}$$

where $\mathbf{p}^{min}$ is the vector containing the minimum values of each element of the input vectors in the dataset, $\mathbf{p}^{max}$ contains the maximum values, $\oslash$ represents an element-by-element (Hadamard) division of two vectors, and $\check{\mathbf{p}}$ is the resulting normalized input vector.

The other normalization method is to scale the inputs so that they have a specified mean and variance – typically 0 and 1. This can be done with the transformation

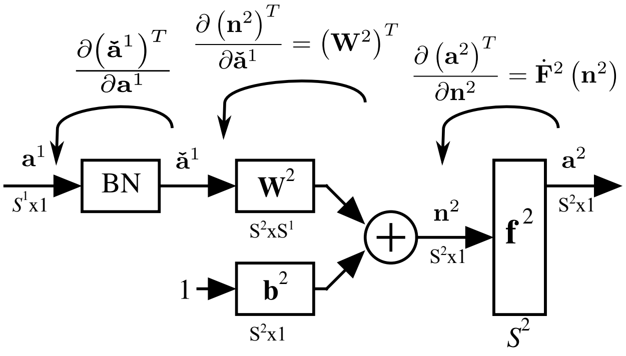
$$\check{\mathbf{p}} = (\mathbf{p} - \mathbf{p}^{mean}) \oslash \mathbf{p}^{std} \tag{3.2}$$

where $\mathbf{p}^{mean}$ is the average of the input vectors in the dataset, and $\mathbf{p}^{std}$ is the vector containing the estimated standard deviations of each element of the input vectors.

These normalizations of the input are helpful for the first layer of the network, and have commonly been used for networks of any size for many years. In 2015 [Ioffe and Szegedy, 2015] proposed that for deep networks the normalizing should be done at each layer in a process they called *batch normalization* (BN). The difficulty is that, unlike the first layer, other layers have inputs (outputs of previous layers) that change during training as the weights and biases are updated. Also, we need to be able to backpropagate across the normalization operations in order to be able to compute the gradient.

[Ioffe and Szegedy, 2015] suggested placing the BN block just before the activation function, but others have also placed it before the weight operation, with similar results.

For example, consider Figure 3.3. Here we have a layer with a batch normalization block (BN) placed at the beginning of the second layer of a multilayer network. This converts the output of Layer 1, $\mathbf{a}^1$, into its normalized version $\breve{\mathbf{a}}^1$.



$\frac{\partial(\breve{\mathbf{a}}^1)^T}{\partial \mathbf{a}^1}$
$\frac{\partial(\mathbf{n}^2)^T}{\partial \breve{\mathbf{a}}^1} = (\mathbf{W}^2)^T$
$\frac{\partial(\mathbf{a}^2)^T}{\partial \mathbf{n}^2} = \dot{\mathbf{F}}^2(\mathbf{n}^2)$
$\mathbf{a}^1$
S^1x1
BN
$\breve{\mathbf{a}}^1$
$\mathbf{W}^2$
S^2xS^1
1
$\mathbf{b}^2$
S^2x1
$\mathbf{n}^2$
S^2x1
$\mathbf{f}^2$
S^2
$\mathbf{a}^2$
S^2x1

Figure 3.3: Backpropagating Across a Layer With Batch Normalization

To backpropagate across this layer, we need to find the derivative across the BN block:

$$\frac{\partial \left(\breve{\mathbf{a}}^1\right)^T}{\partial \mathbf{a}^1}. \tag{3.3}$$

However, the backpropagation required here is different than the type discussed in the previous chapter. The backpropagation in the previous chapter was done one example at a time. This was possible because, for example, the derivative of the layer output $\mathbf{a}^2$ with respect to the net input $\mathbf{n}^2$ for the current example does not depend on the previous example. This is not the case when we are doing batch normalization. The normalization of each example depends on the other examples in the batch. We must backpropagate the entire batch at the same time.

The BN operation begins with the following steps:

$$\mathbf{a}^{mean} = \frac{1}{Q} \sum_{q=1}^{Q} \mathbf{a}_q \tag{3.4}$$

$$\mathbf{a}^{std} = \sqrt{\frac{1}{Q} \sum_{q=1}^{Q} \left(\mathbf{a}_q - \mathbf{a}^{mean}\right)^2 + \varepsilon} \tag{3.5}$$

$$\bar{\mathbf{a}}_q = \left(\mathbf{a}_q - \mathbf{a}^{mean}\right) \oslash \mathbf{a}^{std} \tag{3.6}$$

This is the same as the input normalization of Eq. 3.2, except that in the calculation of $\mathbf{a}^{std}$ the constant ε is added, to ensure that we do not have a divide by zero. (To obtain an unbiased estimate of standard deviation, divide by $(Q-1)$, instead of Q.) Here we can see that the normalized value $\bar{\mathbf{a}}_q$ is dependent on all examples in the batch, which are used in the calculation of $\mathbf{a}^{mean}$ and $\mathbf{a}^{std}$.

In addition to normalizing the data so that the average is 0 and the estimated standard deviation is 1 over that batch, using the equations above, the BN process described in [Ioffe and Szegedy, 2015] adds the following additional operation:

$$\check{\mathbf{a}}_q = \mathbf{w}^{bn} \circ \bar{\mathbf{a}}_q + \mathbf{b}^{bn} \tag{3.7}$$

where $\mathbf{w}^{bn}$ is a weight vector, $\circ$ represents the element-by-element (Hadamard) product and $\mathbf{b}^{bn}$ is a bias vector. The weight in Eq. 3.7 is initially set to all 1's and the bias to zeros, so the operation does not initially have any effect, but it allows the BN operation to learn an appropriate overall scaling and offset. Theoretically, $\mathbf{w}^{bn}$ and $\mathbf{b}^{bn}$ could be incorporated into the following weight and bias ($\mathbf{W}^2$ and $\mathbf{b}^2$ in Figure 3.3), if the BN operation is placed before the weight, but having them separate appears to provide some practical advantage.

In the BN operation, each output depends on all inputs within a batch. Therefore, the backpropagation operation must be done using the entire batch. This means that Figure 3.3 would be replaced with Figure 3.4, where $vec\left(\mathbf{A}\right)$ transforms the matrix into a vector by stacking the columns of the matrix one underneath the other:

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_Q \end{bmatrix} \tag{3.8}$$

$$vec\left(\mathbf{A}\right) = \begin{bmatrix} \mathbf{a}_1^T & \mathbf{a}_2^T & \cdots & \mathbf{a}_Q^T \end{bmatrix}^T \tag{3.9}$$

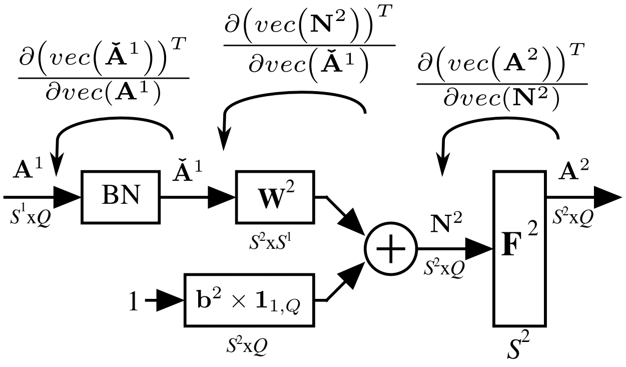

$\frac{\partial(vec(\breve{\mathbf{A}}^1))^T}{\partial vec(\mathbf{A}^1)}$
$\frac{\partial(vec(\mathbf{N}^2))^T}{\partial vec(\breve{\mathbf{A}}^1)}$
$\frac{\partial(vec(\mathbf{A}^2))^T}{\partial vec(\mathbf{N}^2)}$
$\mathbf{A}^1$
S^1xQ
BN
$\breve{\mathbf{A}}^1$
$\mathbf{W}^2$
S^2xS^1
1
$\mathbf{b}^2 \times \mathbf{1}_{1,Q}$
S^2xQ
$\mathbf{N}^2$
S^2xQ
$\mathbf{F}^2$
S^2
$\mathbf{A}^2$
S^2xQ

Figure 3.4: Batch Backpropagating Across a Layer With Batch Normalization

The batch backpropagations across the transfer function and the weight are just a combination of the backpropagations of each example in the batch, as described in Eq. 2.49. This is because of the form of the following Jacobians:

$$\frac{\partial\left(vec\left(\mathbf{N}^2\right)\right)^T}{\partial vec\left(\breve{\mathbf{A}}^1\right)} = \begin{bmatrix} \left(\mathbf{W}^2\right)^T & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \left(\mathbf{W}^2\right)^T & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \left(\mathbf{W}^2\right)^T \end{bmatrix} \tag{3.10}$$

$$\frac{\partial\left(vec\left(\mathbf{A}^2\right)\right)^T}{\partial vec\left(\mathbf{N}^2\right)} = \begin{bmatrix} \dot{\mathbf{F}}^2\left(\mathbf{n}_1^2\right) & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \dot{\mathbf{F}}^2\left(\mathbf{n}_2^2\right) & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \dot{\mathbf{F}}^2\left(\mathbf{n}_Q^2\right) \end{bmatrix} \tag{3.11}$$

However, the derivative across the BN block will not have this block diagonal form, so the backpropagation must be done with the entire batch. It is possible to compute the derivative across BN in one operation, but it is somewhat complex. It is easier to break the operation into simple steps and use the chain rule (backpropagation) to compute it. Figure 3.5 shows the initial BN operation (Eqs. 3.4 to 3.6) in graphical form involving basic steps.

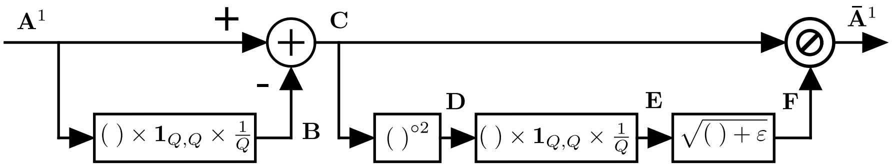

$\mathbf{A}^1$
+
-
$(\,) \times \mathbf{1}_{Q,Q} \times \frac{1}{Q}$
B
C
$(\,)^{\circ 2}$
D
$(\,) \times \mathbf{1}_{Q,Q} \times \frac{1}{Q}$
E
$\sqrt{(\,) + \varepsilon}$
F
$\bar{\mathbf{A}}^1$

Figure 3.5: First Stage of Batch Normalization Operations

The block between $\mathbf{A}^1$ and $\mathbf{B}$ computes the average of the batch (Eq. 3.4) and broadcasts the result to make Q columns (see Solved Problem P3.2). The $\mathbf{C}$ to $\mathbf{F}$ path computes the estimated standard deviation of the batch (Eq. 3.5) and broadcasts the result to make Q columns.

Using the backpropagation process, the derivative across this part of the BN process can be computed using

$$\begin{aligned} \frac{\partial\left(vec\left(\bar{\mathbf{A}}^1\right)\right)^T}{\partial vec\left(\mathbf{A}^1\right)} = & \left[\frac{\partial\left(vec\left(\mathbf{C}\right)\right)^T}{\partial vec\left(\mathbf{A}^1\right)} - \frac{\partial\left(vec\left(\mathbf{B}\right)\right)^T}{\partial vec\left(\mathbf{A}^1\right)}\frac{\partial\left(vec\left(\mathbf{C}\right)\right)^T}{\partial vec\left(\mathbf{B}\right)}\right] \\ & \left[\frac{\partial\left(vec\left(\bar{\mathbf{A}}^1\right)\right)^T}{\partial vec\left(\mathbf{C}\right)} + \frac{\partial\left(vec\left(\mathbf{F}\right)\right)^T}{\partial vec\left(\mathbf{C}\right)}\frac{\partial\left(vec\left(\bar{\mathbf{A}}^1\right)\right)^T}{\partial vec\left(\mathbf{F}\right)}\right] \end{aligned} \tag{3.12}$$

The derivatives across the summation point are simple:

$$\frac{\partial\left(vec\left(\mathbf{C}\right)\right)^T}{\partial vec\left(\mathbf{A}^1\right)} = \mathbf{I}_{S^1 \cdot Q} \tag{3.13}$$

$$\frac{\partial\left(vec\left(\mathbf{C}\right)\right)^T}{\partial vec\left(\mathbf{B}\right)} = -\mathbf{I}_{S^1 \cdot Q} \tag{3.14}$$

where $\mathbf{I}_N$ is an $N \times N$ identity matrix. The averaging operations from $\mathbf{A}^1$ to $\mathbf{B}$ and $\mathbf{D}$ to $\mathbf{E}$ have derivative

$$\frac{\partial\left(vec\left(\mathbf{B}\right)\right)^T}{\partial vec\left(\mathbf{A}^1\right)} = \frac{\partial\left(vec\left(\mathbf{E}\right)\right)^T}{\partial vec\left(\mathbf{D}\right)} = \mathbf{I}_{S^1}^{Q} \times \frac{1}{Q} \tag{3.15}$$

where

$$\mathbf{I}_S^Q = \left.\begin{bmatrix} \mathbf{I}_S & \mathbf{I}_S & \cdots & \mathbf{I}_S \\ \mathbf{I}_S & \mathbf{I}_S & \cdots & \mathbf{I}_S \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{I}_S & \mathbf{I}_S & \cdots & \mathbf{I}_S \end{bmatrix}\right\} Q \cdot S \tag{3.16}$$

The derivative across the $\mathbf{C}$ to $\mathbf{F}$ path, which computes the estimated standard deviation, can be found by backpropagation:

$$\frac{\partial\left(vec\left(\mathbf{F}\right)\right)^T}{\partial vec\left(\mathbf{C}\right)} = \frac{\partial\left(vec\left(\mathbf{D}\right)\right)^T}{\partial vec\left(\mathbf{C}\right)}\frac{\partial\left(vec\left(\mathbf{E}\right)\right)^T}{\partial vec\left(\mathbf{D}\right)}\frac{\partial\left(vec\left(\mathbf{F}\right)\right)^T}{\partial vec\left(\mathbf{E}\right)} \tag{3.17}$$

where

$$\frac{\partial\left(vec\left(\mathbf{D}\right)\right)^T}{\partial vec\left(\mathbf{C}\right)} =2 \cdot diag(vec(\mathbf{C})) \tag{3.18}$$

$$\frac{\partial\left(vec\left(\mathbf{F}\right)\right)^T}{\partial vec\left(\mathbf{E}\right)} = -\frac{1}{2} \cdot diag(vec(\mathbf{E}+\varepsilon))^{-\frac{1}{2}} \tag{3.19}$$

where $diag(\mathbf{x})$ forms a diagonal matrix with the vector $\mathbf{x}$ along the diagonal.

To complete the derivative across the main part of the BN process, using Eq. 3.12, we need only the last two terms (see Solved Problem P3.4):

$$\frac{\partial \left(vec\left(\bar{\mathbf{A}}^1\right)\right)^T}{\partial vec\left(\mathbf{C}\right)} = diag(vec(\mathbf{F}))^{-1} \tag{3.20}$$

$$\frac{\partial \left(vec\left(\bar{\mathbf{A}}^1\right)\right)^T}{\partial vec\left(\mathbf{F}\right)} = -diag\left(vec(\mathbf{C}) \oslash vec(\mathbf{F})^{\circ 2}\right) \tag{3.21}$$

The final stage of the BN process, Eq. 3.7, involves just a weight multiplication and then the addition of a bias. This is the same as a standard layer in a multilayer network, except that the weight matrix is a vector and the weight operation is a Hadamard product. See Solved Problem P.3.1 for a derivation of backpropagation across the Hadamard product.

Inference refers to the process of using the network on new data after the network has been trained.

After training, when the network is being used for inference, it is not reasonable to use the BN block in the same manner that it is used during training. In some cases, only one example (a batch of one) might be applied to the network, and it would not be possible to calculate a standard deviation for a batch of one example. For inference, the BN process should be fixed, which means that fixed values would be used for $\mathbf{a}^{mean}$ and $\mathbf{a}^{std}$. (The weight and bias $\mathbf{w}^{bn}$ and $\mathbf{b}^{bn}$ of Eq. 3.7 would be fixed during inference at their trained values.) One choice would be to use the values obtained from the full training set. However, [Ioffe and Szegedy, 2015] suggests using a smoothed version of the values computed during training, as in

$$\tilde{\mathbf{a}}_k^{mean} = \delta \cdot \tilde{\mathbf{a}}_{k-1}^{mean} + (1-\delta) \cdot \mathbf{a}_k^{mean} \tag{3.22}$$

$$\tilde{\mathbf{a}}_k^{std} = \delta \cdot \tilde{\mathbf{a}}_{k-1}^{std} + (1-\delta) \cdot \mathbf{a}_k^{std} \tag{3.23}$$

where $\mathbf{a}_k^{mean}$ and $\mathbf{a}_k^{std}$ are the mean and standard deviation computed at the k^{th} iteration (minibatch) of training, and $0 < \delta < 1$ determines the amount of smoothing (δ close to one averages over more iterations). At the completion of training, the final $\tilde{\mathbf{a}}^{mean}$ and $\tilde{\mathbf{a}}^{std}$ would be used as fixed values during inference.

It should be noted that batch normalization was developed with sigmoid activations in mind – the idea being to keep activations in the non-saturating region of the sigmoid. However, experiments have shown that it also improves performance with *ReLU* (*poslin*) networks. It has been suggested that this is because it tends to improve the generalization abilities of the network by causing random changes in scaling from minibatch to minibatch. In this way, it may have an effect similar to *dropout*, which is discussed in a later section of this chapter. Another reason for the benefits of BN is provided in [Santurkar et al., 2018], which suggests that BN smooths the performance function surface.

Normalization is one way to minimize the vanishing gradient problem, because it helps to keep the inputs to the activation functions in their non-saturating regions. This can also be done by proper weight initialization. This is because it is the product between the weights and the inputs to the layer that determines the input to the activation function. Next we consider several techniques for weight initialization.

Weight initialization sets the values of the weights and biases that are used in the first iteration of the training algorithm. If the initial weights are too large, the sigmoid activation functions can be saturated, which means that the algorithm would start in flat regions of the performance surface. Small gradients in this region would lead to very slow convergence. This was illustrated in Chapter 12 of NND2.

It was also shown in Chapter 12 of NND2 that the initial weights should not be set to zero, since this is a saddle point in the performance surface, where the gradient would be zero. One solution is to set the weights to small random values. Of course, small is a relative term. The key is that we do not want to saturate the sigmoid activation function. In addition, as we can see from Eq. 2.49, the sensitivity is multiplied by the weights at each stage of the backpropagation process. This means that if the weights are too small, the gradient will become very small for weights in the early layers of a deep network (*vanishing gradient*).

[Nguyen and Widrow, 1990] were the first to suggest setting the initial weights to larger values. As described in Chapter 20 of NND2, the idea was to set the magnitude of the weights in the first layer, containing S^1 neurons, so that the linear region of each sigmoid function covers $1/S^1$ of the range of the input. The biases are then randomly set, so that the center of each sigmoid function falls randomly in the input space. This technique was designed for two-layer networks, but could be extended for multiple layers.

[Glorot and Bengio, 2010] proposed an initialization strategy specifically designed for deep networks with hyperbolic tangent sigmoid activation functions. The idea was to produce weights that would not change the variation in activations as they propagated forward through the network, and at the same time would not change the variation in sensitivities as they propagated backward through the network.

Consider the forward propagation to layer m:

$$\mathbf{a}^m = \mathbf{f}^m\left(\mathbf{W}^m\mathbf{a}^{m-1} + \mathbf{b}^m\right) \tag{3.24}$$

if we are operating in the linear region of the hyperbolic tangent sigmoid function, where the slope is 1, we can write

$$\mathbf{a}^m \simeq \mathbf{W}^m\mathbf{a}^{m-1} + \mathbf{b}^m \tag{3.25}$$

Considering one element of the layer output

$$a_i^m \simeq \sum_{j=1}^{S^{m-1}} w_{i,j}^m a_j^{m-1} + b_i^m \tag{3.26}$$

[Glorot and Bengio, 2010] set the initial biases to zero, so the last term is removed. We also make the assumption that the all activations coming from the previous layer are random, with mean zero and variance $\sigma_a^2(m-1)$ and the initial weights are random values with mean zero and variance σ_w^2. To simplify the analysis, we also assume that the activations and weights are independent. This produces the following result.

$$var\left(a_i^m\right) \equiv \sigma_a^2(m) = S^{m-1}\sigma_w^2\sigma_a^2(m-1) \tag{3.27}$$

Therefore, if we want the activation variance to remain the same from layer to layer (i.e., $\sigma_a^2(m) = \sigma_a^2(m-1)$), we should set

$$\sigma_w^2 = \frac{1}{S^{m-1}} \tag{3.28}$$

Now consider the backpropagation process of Eq. 2.49:

$$\mathbf{s}^{m-1} = \dot{\mathbf{F}}^{m-1}(\mathbf{n}^{m-1})(\mathbf{W}^m)^T\mathbf{s}^m \tag{3.29}$$

If we assume again that we are operating in the linear region of the tangent sigmoid, the derivative will be approximately 1. We can then approximately compute an element of the sensitivity as

$$s_i^{m-1} \simeq \sum_{j=1}^{S^m} w_{j,i}^m s_j^m \tag{3.30}$$

Making the same assumptions concerning independence that were made for the forward propagation, and assuming that the sensitivities at layer m have mean zero and variance $\sigma_s^2(m)$, we can approximate the variance of the sensitivities at layer $m-1$ to be

$$var\left(s_i^{m-1}\right) = \sigma_s^2(m-1) = S^m \sigma_w^2 \sigma_s^2(m) \tag{3.31}$$

If we want the sensitivity variance to remain the same from layer to layer (i.e., $\sigma_s^2(m-1) = \sigma_s^2(m)$), then we should set

$$\sigma_w^2 = \frac{1}{S^m} \tag{3.32}$$

In [Glorot and Bengio, 2010], they suggest combining Eqs. 3.28 and 3.32 so that the variance of the weights should be

$$\sigma_w^2 = \frac{2}{S^{m-1} + S^m} \tag{3.33}$$

So, in *Xavier initialization*, the initial weights in layer m are set as normally distributed random numbers with mean zero and variance set by Eq. 3.33. The biases are initially set to zero.

Xavier initialization was designed for the hyperbolic tangent sigmoid activation function. If the *ReLU* (*poslin*) activation function is used, then a slight adjustment is made to the variance of the weights. In [He et al., 2015], they show that, since the *poslin* function is zero for half of its domain, the variance is cut in half as a signal is passed through that activation function, so the variance of the weights should be doubled to maintain the variance of the activations from layer to layer. The resulting *Kaiming initialization* uses a variance for the initial weights of

$$\sigma_w^2 = \frac{2}{S^{m-1}} \tag{3.34}$$

The combination of normalization and proper weight initialization are helpful in preventing a vanishing gradient.

To experiment with the use of different normalization and initialization effects, use the Deep Learning Demonstration Scaling (**dl4sc**).

To experiment with different initialization schemes and how they affect the range of layer outputs as the number of layers increase, use the Deep Learning Demonstration Initialization Effect (**dl4ie**).

Gradient clipping is another technique that helps to speed up convergence. First formally proposed in [Pascanu et al., 2013] for training recurrent networks (which were discussed in Chapter 14 of NND2 and will be discussed in a later chapter of this volume), it involves limiting the magnitude of the gradient. We previously discussed the vanishing gradient problem, in which the gradients can get smaller as the number of layers get large. This is because, as we can see from Eq. 2.49, the sensitivity is multiplied by the weights at each stage of the backpropagation process. This means that if the weights are too small, the gradient will become very small for weights in the early layers of a deep network. For a similar reason, if the weights are too large, the gradients can become very large (*exploding gradient*). This is most likely to occur in recurrent networks, as we saw in Chapter 14 of NND2, and as we will see in a later chapter of this volume.

[Pascanu et al., 2013] propose that if the norm of the gradient goes beyond a specified threshold that the gradient be normalized so that the norm is equal to the threshold. In this way, the gradient search algorithm moves in the negative gradient direction, but with a smaller step size. This would be equivalent to reducing the learning rate whenever the gradient becomes too large. Another option would be to limit the individual elements of the gradient to a specific absolute value, although this would change the direction of search. Both of these techniques are referred to as *gradient clipping*.

Generalization

Although the previous chapter focused on network training as optimizing a performance function on a specified training set, the true objective of network training is to produce a network that will perform well on future data that it has not seen before. This ability to generalize was investigated extensively in Chapter 13 of NND2, which also surveyed techniques for improving the generalization ability of trained networks.

The two main approaches to improving generalization that were covered in Chapter 13 of NND2 were *early stopping* and regularization. The idea behind early stopping is that, as the network trains, it is using more and more of its weights, and all of the weights are fully used when the network reaches a minimum of the performance surface. By increasing the number of iterations of training, we are increasing the complexity of the resulting network. The more complex the network is, the more likely it is to overfit the training data and, therefore, to perform poorly on new data that it has not seen. The key question is when to stop the training.

For early stopping, a portion of the training data is set aside as a *validation set*. The performance on the validation set is monitored during training, but it is not used to compute the gradient. An increase in the validation performance indicates that overfitting is occurring, so the weights that produce the minimum validation performance are used as the final trained weights. See Chapter 13 of NND2 for a more complete description and a software demonstration of early stopping.

At this point, we should clarify that when early stopping is used, the full data set is divided into three components: training, validation and testing. The *test set* is removed from the full data set before training. It is not used in any way during the training process, even for selecting between potential networks. After training is complete, the test set is applied to the network, and the resulting performance is used to measure the generalization ability of the network. One of the most common mistakes in neural network training is misuse of the testing set. It must never be used in any way until the final network has been trained. Otherwise, the performance on the test set will be overly optimistic about future performance.

Regularization, also described in Chapter 13 of NND2, is a method which modifies the performance function by adding a term that penalizes network complexity. By penalizing complexity, the training will produce a network that will be better able to generalize. The most common penalty is the sum squared weights. The addition of this penalty to the mean square error performance function would be

$$F(\mathbf{x}) = \frac{1}{QS^M} \sum_{q=1}^{Q} \sum_{i=1}^{S^M} \left(t_{i,q} - a_{i,q}\right)^2 + \rho \sum_{i=1}^{N} x_i^2 \tag{3.35}$$

where $\mathbf{x}$ is the vector containing the weights of the network. This is often referred to as L2 regularization, because it involves the square of the weights. L1 regularization uses the absolute value of the weights.

The difficulty in using regularization is choosing the regularization coefficient ρ. If ρ is chosen too large, the network will underfit, and if ρ is chosen too small, the network will overfit. One technique for choosing ρ is to minimize the unregularized performance on a validation set. Another technique is to use Bayesian regularization to determine ρ. This method is described in Chapter 13 of NND2.

A method for improving generalization that is very efficient, and therefore well-suited to deep networks, is called *dropout*, which was introduced in [Srivastava et al., 2014]. Dropout can be applied to any layer of a network. If it is applied to layer m, then at each iteration of training, each neuron in that layer is activated with a given probability – say ψ_m. If it is not active, then that neuron and its incoming and outgoing connections are removed from the network for both the forward propagation of activations and the backward propagation of sensitivities. (Row i of $\mathbf{W}^m$ and column i of $\mathbf{W}^{m+1}$ are removed if neuron a_i^m is not active). At each iteration, different neurons are randomly activated. Because a given neuron is only available for ψ_m of the iterations, and a different minibatch is used in each iteration, it is unlikely that the network will overfit by memorizing the training set. This is why dropout improves generalization.

When the network is used for inference (after training is complete), the weights need to be scaled to account for the fact that they were not used during all iterations of training. If a given neuron a_i^m in layer m is activated with probability ψ_m during training, then the outgoing weights for that neuron (column i of $\mathbf{W}^{m+1}$) are multiplied by ψ_m during inference. Assuming that all neurons in layer m are activated with probability ψ_m, then the weight matrix $\mathbf{W}^{m+1}$ is multiplied by ψ_m for inference.

We should note that there is something of a conflict between dropout and the batch normalization process that we described in an earlier section (see [Li et al., 2018]). The standard form of dropout is not normally used in combination with BN. However, [Ioffe and Szegedy, 2015] state that in their experiments dropout is often not needed, because BN improves the generalization ability of networks when used alone. The suggested reason for this is that BN introduces a transformation between layers that changes with each iteration (and minibatch), creating a kind of noise in the training that reduces the chances for overfitting.

To experiment with dropout and how it prevents overfitting, use the Deep Learning Demonstration Dropout (**`dl4do`**).

Epilogue

The theoretical principles behind the training of neural networks with an arbitrary number of layers have been widely known since the late 1980s. However, deep networks were not commonly used until techniques were developed for reducing training times. This chapter described several of those techniques.

One way to speed up convergence is to provide the optimization algorithm with a suitable starting point. If sigmoid activation functions are used, then it is important that the initial weights are not so large that the sigmoids are regularly saturated. In this case, the derivatives will be very small, causing gradients to be small and training to be slow. On the other hand, if the weights, which multiply sensitivities during the backpropagation process, are too small, then this will also cause the gradients to be small. The trick is to get into the Goldilocks zone, where the initial neuron activations are neither too large or too small.

The activations are dependent on the inputs to a layer and the layer weights. The inputs to the layer can be brought into the Goldilocks zone through batch normalization, in which the inputs are transformed to have a mean of 0 and a standard deviation of 1. The initial weights can be brought into the Goldilocks zone using Xavier initialization, where they are assigned random values with standard deviation determined by the number of elements of the layer input and layer output. With this choice, activations and sensitivities maintain a similar level throughout the layers of the network.

The main purpose of neural network training is to produce networks that will perform well on new data that were not seen during training – we want the networks to generalize. Dropout is a method to improve network generalization that randomly removes a certain percentage of neurons during each iteration of the training process. Networks are less likely to memorize the training examples, since different neurons may see different minibatches with different frequencies.

Now that we have covered network training procedures, the next step is to implement the algorithms in software and verify their performance on practical data sets. That is the objective of the next couple of chapters.

Further Reading

[Ioffe and Szegedy, 2015] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015

This is the original description of the batch normalization algorithm. It suggests using batch normalization because of covariate shift, in which the distribution of data changes. Others have since questioned that mechanism.

[Santurkar et al., 2018] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization?(no, it is not about internal covariate shift). *arXiv preprint arXiv:1805.11604*, 2018

Batch normalization improves training performance and produces networks that generalize better because it smooths the performance surface.

[Nguyen and Widrow, 1990] Derrick Nguyen and Bernard Widrow. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. In *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, pages 21–26. IEEE, 1990

Describes a method for weight initialization. The initial weights and biases are set so that the sigmoid activations are in their linear range over some specified proportion of the input space.

[Glorot and Bengio, 2010] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010

TDescribes a method for initializing the weights of deep networks so that the activations going forward and the gradients propagating backward maintain a reasonable level of variation. The method is usually referred to as Xavier initialization.

[He et al., 2015] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015

Xavier initialization is based on sigmoid activation functions. This paper describes an alternative form of initialization for ReLU networks. It also describes some activation functions that are variations on the ReLU, with adjustable nonzero slopes for negative inputs.

Summary of Results

Normalize to [-1 1]

$$\check{\mathbf{p}} = \left(\mathbf{p} - \mathbf{p}^{min}\right) \oslash \left(\mathbf{p}^{max} - \mathbf{p}^{min}\right) - 1$$

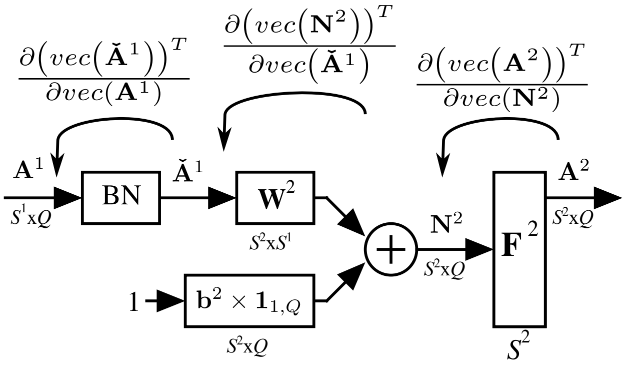
Normalize to Mean 0 and Variance 1

$$\check{\mathbf{p}} = (\mathbf{p} - \mathbf{p}^{mean}) \oslash \mathbf{p}^{std}$$

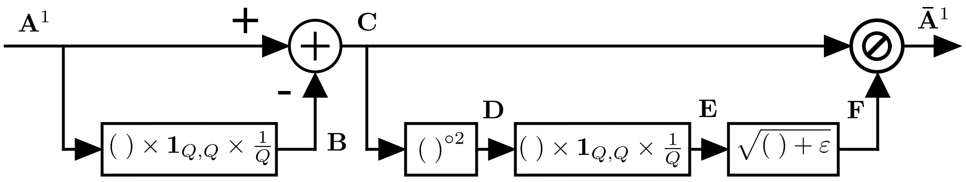
Batch Normalization

$$\begin{aligned}
\mathbf{a}^{mean} &= \frac{1}{Q}\sum_{q=1}^{Q} \mathbf{a}_q \\
\mathbf{a}^{std} &= \sqrt{\frac{1}{Q}\sum_{q=1}^{Q} \left(\mathbf{a}_q - \mathbf{a}^{mean}\right)^2 + \varepsilon} \\
\bar{\mathbf{a}}_q &= \left(\mathbf{a}_q - \mathbf{a}^{mean}\right) \oslash \mathbf{a} \\
\check{\mathbf{a}}_q &= \mathbf{w}^{bn} \circ \bar{\mathbf{a}}_q + \mathbf{b}^{bn}
\end{aligned}$$

Batch Backpropagation Across Layer with BN


$\frac{\partial\left(vec\left(\check{\mathbf{A}}^1\right)\right)^T}{\partial vec(\mathbf{A}^1)}$
$\frac{\partial\left(vec\left(\mathbf{N}^2\right)\right)^T}{\partial vec\left(\check{\mathbf{A}}^1\right)}$
$\frac{\partial\left(vec\left(\mathbf{A}^2\right)\right)^T}{\partial vec(\mathbf{N}^2)}$
$\mathbf{A}^1$
S^1xQ
BN
$\check{\mathbf{A}}^1$
$\mathbf{W}^2$
S^2xS^1
1
$\mathbf{b}^2 \times \mathbf{1}_{1,Q}$
S^2xQ
$\mathbf{N}^2$
S^2xQ
$\mathbf{F}^2$
S^2
$\mathbf{A}^2$
S^2xQ

First Stage of Batch Normalization Operations


$\mathbf{A}^1$
+
-
$\mathbf{C}$
$\bar{\mathbf{A}}^1$
$(\,) \times \mathbf{1}_{Q,Q} \times \frac{1}{Q}$
$\mathbf{B}$
$(\,)^{\circ 2}$
$\mathbf{D}$
$(\,) \times \mathbf{1}_{Q,Q} \times \frac{1}{Q}$
$\mathbf{E}$
$\sqrt{(\,)+\varepsilon}$
$\mathbf{F}$

Backpropagating Across BN Stage

$$\begin{aligned}\frac{\partial \left(vec\left(\bar{\mathbf{A}}^1\right)\right)^T}{\partial vec\left(\mathbf{A}^1\right)} &= \left[\frac{\partial \left(vec\left(\mathbf{C}\right)\right)^T}{\partial vec\left(\mathbf{A}^1\right)} - \frac{\partial \left(vec\left(\mathbf{B}\right)\right)^T}{\partial vec\left(\mathbf{A}^1\right)}\frac{\partial \left(vec\left(\mathbf{C}\right)\right)^T}{\partial vec\left(\mathbf{B}\right)}\right] \\ &\quad \left[\frac{\partial \left(vec\left(\bar{\mathbf{A}}^1\right)\right)^T}{\partial vec\left(\mathbf{C}\right)} + \frac{\partial \left(vec\left(\mathbf{F}\right)\right)^T}{\partial vec\left(\mathbf{C}\right)}\frac{\partial \left(vec\left(\bar{\mathbf{A}}^1\right)\right)^T}{\partial vec\left(\mathbf{F}\right)}\right]\end{aligned}$$

Component Derivatives

$$\begin{aligned}
\frac{\partial \left(vec\left(\mathbf{C}\right)\right)^T}{\partial vec\left(\mathbf{A}^1\right)} &= \mathbf{I}_{S^1 \cdot Q} \\
\frac{\partial \left(vec\left(\mathbf{C}\right)\right)^T}{\partial vec\left(\mathbf{B}\right)} &= -\mathbf{I}_{S^1 \cdot Q} \\
\frac{\partial \left(vec\left(\mathbf{B}\right)\right)^T}{\partial vec\left(\mathbf{A}^1\right)} &= \frac{\partial \left(vec\left(\mathbf{E}\right)\right)^T}{\partial vec\left(\mathbf{D}\right)} = \mathbf{I}_{S^1}^{Q} \times \frac{1}{Q} \\
\mathbf{I}_S^Q &= \left.\begin{bmatrix} \mathbf{I}_S & \mathbf{I}_S & \cdots & \mathbf{I}_S \\ \mathbf{I}_S & \mathbf{I}_S & \cdots & \mathbf{I}_S \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{I}_S & \mathbf{I}_S & \cdots & \mathbf{I}_S \end{bmatrix}\right\} Q \cdot S \\
\frac{\partial \left(vec\left(\mathbf{F}\right)\right)^T}{\partial vec\left(\mathbf{C}\right)} &= \frac{\partial \left(vec\left(\mathbf{D}\right)\right)^T}{\partial vec\left(\mathbf{C}\right)}\frac{\partial \left(vec\left(\mathbf{E}\right)\right)^T}{\partial vec\left(\mathbf{D}\right)}\frac{\partial \left(vec\left(\mathbf{F}\right)\right)^T}{\partial vec\left(\mathbf{E}\right)} \\
\frac{\partial \left(vec\left(\mathbf{D}\right)\right)^T}{\partial vec\left(\mathbf{C}\right)} &= 2 \cdot diag(vec(\mathbf{C})) \\
\frac{\partial \left(vec\left(\mathbf{F}\right)\right)^T}{\partial vec\left(\mathbf{E}\right)} &= -\frac{1}{2} \cdot diag(vec(\mathbf{E} + \varepsilon))^{-\frac{1}{2}} \\
\frac{\partial \left(vec\left(\bar{\mathbf{A}}^1\right)\right)^T}{\partial vec\left(\mathbf{C}\right)} &= diag(vec(\mathbf{F}))^{-1} \\
\frac{\partial \left(vec\left(\bar{\mathbf{A}}^1\right)\right)^T}{\partial vec\left(\mathbf{F}\right)} &= -diag\left(vec(\mathbf{C}) \oslash vec(\mathbf{F})^{\circ 2}\right)
\end{aligned}$$

Smoothed Mean and Standard Deviation for BN Inference

$$\begin{aligned}
\tilde{\mathbf{a}}_k^{mean} &= \delta \cdot \tilde{\mathbf{a}}_{k-1}^{mean} + (1-\delta) \cdot \mathbf{a}_k^{mean} \\
\tilde{\mathbf{a}}_k^{std} &= \delta \cdot \tilde{\mathbf{a}}_{k-1}^{std} + (1-\delta) \cdot \mathbf{a}_k^{std}
\end{aligned}$$

Weight Variance for Xavier Initialization

$$\sigma_w^2 = \frac{2}{S^{m-1} + S^m}$$

Weight Variance for Kaiming Initialization

$$\sigma_w^2 = \frac{2}{S^{m-1}}$$

Solved Problems

P3.1 **The final step of the batch normalization process is Eq. 3.7. This has the appearance of a standard layer, with a linear transfer function. The only difference is that instead of standard matrix multiplication, it uses the Hadamard product. How should you backpropagate across the Hadamard product?**

Repeating Eq. 3.7 here, we have

$$\check{\mathbf{a}}_q = \mathbf{w}^{bn} \circ \bar{\mathbf{a}}_q + \mathbf{b}^{bn}$$

This does appear to be a standard linear layer, with Hadamard product as the weight function. From Eq. 2.45, the sensitivity update across a layer would be

$$\mathbf{s}^m = \frac{\partial \left(\mathbf{a}^m\right)^T}{\partial \mathbf{n}^m} \frac{\partial \left(\mathbf{z}^{m+1}\right)^T}{\partial \mathbf{a}^m} \frac{\partial \left(\mathbf{n}^{m+1}\right)^T}{\partial \mathbf{z}^{m+1}} \mathbf{s}^{m+1}$$

The term that is different for this layer is the weight function. We just need to compute

$$\frac{\partial \left(\mathbf{z}^{m+1}\right)^T}{\partial \mathbf{a}^m}$$

For the Hadamard product, the weight function is

$$\begin{aligned} \mathbf{z}^{m+1} &= \mathbf{w}^{m+1} \circ \mathbf{a}^m \\ z_i^{m+1} &= w_i^{m+1} a_i^m \end{aligned}$$

The elements of the Jacobian would be

$$\left[\frac{\partial \left(\mathbf{z}^{m+1}\right)^T}{\partial \mathbf{a}^m}\right]_{i,j} = \frac{\partial z_j^{m+1}}{\partial a_i^m} = \begin{cases} w_i^{m+1} & i = j \\ 0 & i \neq j \end{cases}$$

or, in matrix form

$$\frac{\partial \left(\mathbf{z}^{m+1}\right)^T}{\partial \mathbf{a}^m} = diag\left(\mathbf{w}^{m+1}\right)$$

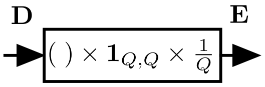
The backpropagation across the layer would then reduce to

$$\mathbf{s}^m = \dot{\mathbf{F}}^m\left(\mathbf{n}^m\right) diag\left(\mathbf{w}^{m+1}\right) \mathbf{s}^{m+1}$$

which is the same as

$$\mathbf{s}^{m} = \dot{\mathbf{F}}^{m}\left(\mathbf{n}^{m}\right)\mathbf{w}^{m+1} \circ \mathbf{s}^{m+1}$$

P3.2 **The first stage of the batch normalization process is displayed graphically in Figure 3.5. There are two blocks in this process that perform the same averaging operation: $\mathbf{A}^{1}$ to B and D to E. Demonstrate how the block performs the averaging operation, and show how the derivative is given by Eq. 3.15.**



The calculation can be written in matrix form as

$$\mathbf{E} = \mathbf{D} \times \mathbf{1}_{Q,Q} \times \frac{1}{Q}$$

$$\mathbf{E} = \begin{bmatrix} d_{1,1} & d_{1,2} & \cdots & d_{1,Q} \\ d_{2,1} & d_{2,2} & \cdots & d_{2,Q} \\ \vdots & \vdots & \ddots & \vdots \\ d_{S^1,1} & d_{S^1,2} & \cdots & d_{S^1,Q} \end{bmatrix} \times \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix} \times \frac{1}{Q}$$

$$= \begin{bmatrix} \frac{1}{Q}\sum_{q=1}^{Q} d_{1,q} & \frac{1}{Q}\sum_{q=1}^{Q} d_{1,q} & \cdots & \frac{1}{Q}\sum_{q=1}^{Q} d_{1,q} \\ \frac{1}{Q}\sum_{q=1}^{Q} d_{2,q} & \frac{1}{Q}\sum_{q=1}^{Q} d_{2,q} & \cdots & \frac{1}{Q}\sum_{q=1}^{Q} d_{2,q} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{Q}\sum_{q=1}^{Q} d_{S^1,q} & \frac{1}{Q}\sum_{q=1}^{Q} d_{S^1,q} & \cdots & \frac{1}{Q}\sum_{q=1}^{Q} d_{S^1,q} \end{bmatrix}$$

$$e_{i,q} = \frac{1}{Q}\sum_{q'=1}^{Q} d_{i,q'}$$

Therefore, each column of $\mathbf{E}$ contains the average of the columns of $\mathbf{D}$. The derivative we want to calculate is

$$\frac{\partial\left(vec\left(\mathbf{E}\right)\right)^{T}}{\partial vec\left(\mathbf{D}\right)}$$

The elements of this Jacobian have the form

$$\frac{\partial e_{i,q}}{\partial d_{i',q'}} = \begin{cases} \frac{1}{Q} & i' = i \\ 0 & i' \neq i \end{cases}$$

which produces Eq. 3.15.

P3.3 **The normalization process changes the mean and variance of incoming data. Suppose that it was known that the input vector p had a mean μ_p and covariance matrix $\mathbf{Q}_p$. Let the normalization process have the form $\bar{\mathbf{p}} = \mathbf{W}\mathbf{p} + \mathbf{b}$.**

i. **Find the mean of $\bar{\mathbf{p}}$, and find the conditions on W and b so that the mean of $\bar{\mathbf{p}}$ will be zero.**

ii. **Find the covariance matrix of $\bar{\mathbf{p}}$, and find a W so that the elements of $\bar{\mathbf{p}}$ have variance of 1 and are uncorrelated with each other.**

i. The mean of $\bar{\mathbf{p}}$ can be written

$$E\left[\bar{\mathbf{p}}\right] = \mu_{\bar{p}} = E\left[\mathbf{W}\mathbf{p} + \mathbf{b}\right] = \mathbf{W}\boldsymbol{\mu}_p + \mathbf{b}$$

Therefore, to produce a zero mean result, we should choose

$$\mathbf{b} = -\mathbf{W}\boldsymbol{\mu}_p$$

ii. The covariance matrix for $\bar{\mathbf{p}}$ can be written

$$cov\left\{\bar{\mathbf{p}}\right\} = \mathbf{Q}_{\bar{p}} = E\left[\left(\bar{\mathbf{p}} - \mu_{\bar{p}}\right)\left(\bar{\mathbf{p}} - \mu_{\bar{p}}\right)^T\right]$$

We can write

$$\bar{\mathbf{p}} - \mu_{\bar{p}} = \mathbf{W}\mathbf{p} + \mathbf{b} - \mathbf{W}\boldsymbol{\mu}_p - \mathbf{b} = \mathbf{W}\left(\mathbf{p} - \mu_p\right)$$

Substituting into the earlier equation, we have

$$\mathbf{Q}_{\bar{p}} = E\left[\mathbf{W}\left(\mathbf{p} - \mu_p\right)\left(\mathbf{p} - \mu_p\right)^T \mathbf{W}^T\right] = \mathbf{W}\mathbf{Q}_p\mathbf{W}^T$$

From Chapter 8 of NND2, we know that the eigenvectors of a symmetric matrix $\mathbf{Q}$ are orthogonal, and that $\mathbf{Q}$ can be diagonalized using

$$\Lambda = \mathbf{B}^T\mathbf{Q}\mathbf{B} = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & \lambda_R \end{bmatrix}$$

where

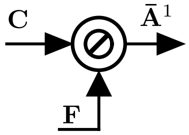
$$\mathbf{B} = \begin{bmatrix} \mathbf{z}_1 & \mathbf{z}_2 & \cdots & \mathbf{z}_R \end{bmatrix}$$

is the matrix whose columns are the eigenvectors of $\mathbf{Q}$, and the λ_i are the eigenvalues of $\mathbf{Q}$.

From this result, if we choose the transformation matrix $\mathbf{W}$ to be $\mathbf{B}^T$, where the columns of $\mathbf{B}$ are the eigenvectors of $\mathbf{Q}_p$, then we would have

$$\mathbf{Q}_{\bar{p}} = \mathbf{W}\mathbf{Q}_p\mathbf{W}^T = \mathbf{B}^T\mathbf{Q}_p\mathbf{B} = \Lambda$$

Because the covariance matrix is diagonal, this means that the elements of $\bar{\mathbf{p}}$ are uncorrelated with each other. However, the variances of the elements are equal to the eigenvalues λ_i. If we want the variances to equal 1, we need to multiply the columns of $\mathbf{B}$ by $1/\sqrt{\lambda_i}$, before performing the diagonalization operation.



P3.4 **For the first stage of the batch normalization process, show how the derivative across the F to $\bar{\mathbf{A}}^1$ path is given by Eq. 3.21.**

The calculation performed by the Hadamard division is

$$\bar{\mathbf{A}}^1 = \mathbf{C} \oslash \mathbf{F}$$
$$\bar{a}^1_{i,j} = \frac{c_{i,j}}{f_{i,j}}$$

Taking the derivative across the $\mathbf{F}$ to $\bar{\mathbf{A}}^1$ path, we have

$$\frac{\partial \bar{a}^1_{i,q}}{\partial f_{i',q'}} = \begin{cases} -\frac{c_{i,q}}{\left(f_{i,q}\right)^2} & i = i', q = q' \\ 0 & else \end{cases}$$

This is the same as Eq. 3.21.

Exercises

E3.1 Assume that a scalar random variable p has a uniform distribution on the interval $[0, 1]$. We transform p to the variable a using $a = wp + b$.

i. Find the values of w and b so that a has mean 0 and variance 1.

ii. Sketch the density function of a. Is is possible to adjust w and b so that a has a Gaussian density? Explain.

E3.2 Suppose that the random variable p has mean 1 and variance 4. Define a second random variable $a = wp + b$. Find the values for w and b so that a has mean 0 and variance 1.

E3.3 Consider a one-layer network with two neurons and a linear activation function. The input vector $\mathbf{p}$ has mean and covariance matrix given by

$$\boldsymbol{\mu}_p = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$\mathbf{Q}_p = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}$$

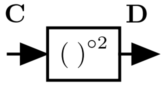
i. Write out an expression for the mean and covariance matrix for the network output $\mathbf{a}$, as a function of the network weight matrix $\mathbf{W}$ and bias vector $\mathbf{b}$.

ii. Find a weight matrix $\mathbf{W}$ so that the network outputs will be uncorrelated and with a variance of 1.

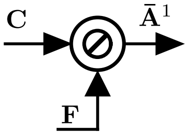
iii. Find the bias vector $\mathbf{b}$ so that the network outputs will have zero mean.

E3.4 If w has variance σ_w^2, b has variance σ_b^2 and p has variance σ_b^2, what is the variance of $a = wp + b$? Assume that w, b and p are mutually uncorrelated.

E3.5 The first stage of the batch normalization process is displayed graphically in Figure 3.5. The **C** to **D** block performs an element-by-element squaring operation. Show how the derivative across this block is given by Eq. 3.18.



E3.6 For the first stage of the batch normalization process, show how the derivative across the $\mathbf{C}$ to $\bar{\mathbf{A}}^1$ path is given by Eq. 3.20


C
$\bar{\mathbf{A}}^1$
F