C H A P T E R 12

Neural Networks

12.1 UNITS AND CLASSIFICATION

We will build complex classification systems out of simple units. A **unit** takes a vector $\mathbf{x}$ of inputs and uses a vector $\mathbf{w}$ of parameters (known as the **weights**), a scalar b (known as the **bias**), and a nonlinear function F to form its output, which is

$$F(\mathbf{w}^T\mathbf{x} + b).$$

Over the years, a wide variety of nonlinear functions have been tried. Current best practice is to use the **RELU** (for rectified linear unit), where

$$F(u) = \mathsf{max}\,(0, u).$$

For example, if $\mathbf{x}$ was a point on the plane, then a single unit would represent a line, chosen by the choice of $\mathbf{w}$ and b. The output for all points on one side of the line would be zero. The output for points on the other side would be a positive number that is larger for points that are further from the line.

Units are sometimes referred to as **neurons**, and there is a large and rather misty body of vague speculative analogy linking devices built out of units to neuroscience. I deprecate this practice; what we are doing here is quite useful and interesting enough to stand on its own without invoking biological authority. Also, if you want to see a real neuroscientist laugh, explain to them how your neural network is really based on some gobbet of brain tissue or other.

12.1.1 Building a Classifier out of Units: The Cost Function

We will build a multiclass classifier out of units by modelling the class posterior probabilities using the outputs of the units. Each class will get the output of a single unit. Write o_i for the output of the i'th unit, and θ for all the parameters in all the units. We will organize these units into a vector $\mathbf{o}$, whose i'th component is o_i. We want to use that unit to model the probability that the input is of class j, which I will write $p(\text{class} = j|\mathbf{x}, \theta)$. To build this model, I will use the **softmax function**. This is a function that takes a C dimensional vector and returns a C dimensional vector. I will write $\mathbf{s}(\mathbf{u})$ for the softmax function, and the dimension C will always be the number of classes. We have

$$\mathbf{s}(\mathbf{u}) = \left(\frac{1}{\sum_k e^{u_k}}\right)\begin{bmatrix} e^{u_1} \\ e^{u_2} \\ \dots \\ e^{u_C} \end{bmatrix}$$

(recall u_i is the i'th component of $\mathbf{u}$). We then use the model

$$p(\text{class} = i|\mathbf{x}, \theta) = s_i(\mathbf{o}(\mathbf{x}, \theta)).$$

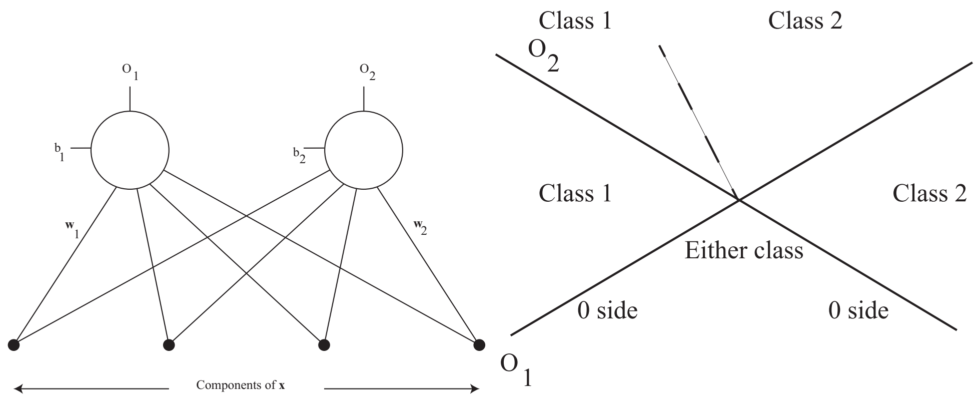


FIGURE 12.1: *On the* **left**, *two units observing an input vector, and providing outputs. On the* **right**, *the decision boundary for two units classifying a point on the plane into one of two classes. The angle of the dashed line depends on the magnitudes of* $\mathbf{w}_1$ *and* $\mathbf{w}_2$.

Notice that this expression passes important tests for a probability model. Each value is between 0 and 1, and the sum over classes is 1.

In this form, the classifier is not super interesting. For example, imagine that the features $\mathbf{x}$ are points on the plane, and we have two classes. Then we have two units, one for each class. There is a line corresponding to each unit; on one side of the line, the unit produces a zero, and on the other side, the unit produces a positive number that increases as with perpendicular distance from the line. We can get a sense of what the decision boundary will be like from this. When a point is on the 0 side of both lines, the class probabilities will be equal (and so both $\frac{1}{2}$ – two classes, remember). When a point is on the positive side of the i'th line, but the zero side of the other, the class probability for class i will be

$$\frac{e^{o_i(\mathbf{x},\theta)}}{1+e^{o_i(\mathbf{x},\theta)}},$$

and the point will always be classified in the i'th class (remember, $o_i \geq 0$). Finally, when a point is on the positive side of both lines, the classifier boils down to choosing the i that has the largest value of $o_i(\mathbf{x}, \theta)$. All this leads to the decision boundary shown in figure **??**. Notice that this is piecewise linear, and somewhat more complex than the boundary of an SVM. It's quite helpful to try and draw what would happen for three or more classes with $\mathbf{x}$ a 2D point.

12.1.2 Building a Classifier out of Units: Strategy

The essential difficulty here is to choose θ that results in the best behavior. We will do so by writing a cost function that estimates the error rate of the classification, then choosing a value $\hat{\theta}$ that minimises that function. We have a set of N examples $\mathbf{x}_i$ and for each example we know the class. There are a total of C classes. We encode the class of an example using a **one hot** vector $\mathbf{y}_i$, which is C dimensional.

If the i'th example is from class j, then the j'th component of $\mathbf{y}_i$ is 1, and all other components in the vector are 0. I will write y_{ij} for the j'th component of $\mathbf{y}_i$.

A natural cost function looks at the log likelihood of the data under the probability model produced from the outputs of the units. If the i'th example is from class j, we would like $-\log p(\text{class} = j|\mathbf{x}_i, \theta)$ to be small (notice the sign here; it's usual to minimize negative log likelihood). I will write $\log \mathbf{s}$ to mean the vector whose components are the logarithms of the components of $\mathbf{s}$. This yields a loss function

$$\frac{1}{N} \sum_{i \in \text{data}} \left[\sum_{j \in \text{classes}} \left\{ -\mathbf{y}_i^T \log \mathbf{s}(\mathbf{o}(\mathbf{x}_i, \theta)) \right\} \right].$$

Notice that this loss function is written in a clean way that may lead to a poor implementation. I have used the y_{ij} values as "switches", as in the discussion of EM. This leads to clean notation, but hides fairly obvious computational efficiencies (when taking the gradient, you need to deal with only one term in the sum over classes). As in the case of the linear SVM (section 10.1), we would like to achieve a low cost with a "small" θ, and so form an overall cost function that will have loss and penalty terms.

There are a variety of possible penalties. For now, we will penalize large sets of weights, but we'll look at other possibilities below. Remember, we have C units (one per class) and so there are C distinct sets of weights. Write the weights for the u'th unit $\mathbf{w}_u$. Our penalty becomes

$$\sum_{u \in \text{units}} \mathbf{w}_u^T \mathbf{w}_u.$$

As in the case of the linear SVM (section 10.1), we write λ for a weight applied to the penalty. Our cost function is then

$$S(\theta, \mathbf{x}; \lambda) = \underbrace{\frac{1}{N} \sum_{i \in \text{data}} \left[\sum_{j \in \text{classes}} \left\{ -\mathbf{y}_i^T \log \mathbf{s}(\mathbf{o}(\mathbf{x}_i, \theta)) \right\} \right]}_{\text{(misclassification loss)}} + \underbrace{\frac{\lambda}{2} \sum_{u \in \text{units}} \mathbf{w}_u^T \mathbf{w}_u}_{\text{(penalty)}}$$

12.1.3 Building a Classifier out of Units: Training

I have described a simple classifier built out of units. We must now train this classifier, by choosing a value of θ that results in a small loss. It may be quite hard to get the true minimum, and we may need to settle for a small value. We use stochastic gradient descent, because we have seen it before; because it is effective; and because it is the algorithm of choice when training more complex classifiers built out of units.

For the SVM, we selected one example at random, computed the gradient at that example, updated the parameters, and went again. For neural nets, it is more usual to use **minibatch training**, where we select a subset of the data uniformly and at random, compute a gradient using that subset, update and go again. This is because in the best implementations many operations are vectorized, and using a minibatch can provide a gradient estimate that is clearly better than that obtained using only one example, but doesn't take longer to compute. The size of the minibatch is usually determined by memory or architectural considerations. It is often a power of two, for this reason.

Now imagine we have chosen a minibatch of M examples. We must compute the gradient of the cost function. This is mainly an exercise in notation, but there's a lot of notation. Write θ_u for a vector containing all the parameters for the u'th unit, so that $\theta_u = [\mathbf{w}_u, b_u]^T$. Recall $s_k(\mathbf{o}(\mathbf{x}_i, \theta_k))$ is the output of the softmax function for the k'th unit for input $\mathbf{x}_i$. This represents the probability that example i is of class k under the current model. Then we must compute

$$\nabla_{\theta_u} \frac{1}{M} \sum_{i \in \text{minibatch}} \left[\left\{-\mathbf{y}_i^T \log \mathbf{s}(\mathbf{o}(\mathbf{x}_i, \theta))\right\}\right] + \frac{\lambda}{2} \sum_{j \in \text{classes}} \mathbf{w}_j^T \mathbf{w}_j.$$

The gradient is easily computed using the chain rule. The term

$$\frac{\lambda}{2} \sum_{j \in \text{classes}} \mathbf{w}_j^T \mathbf{w}_j$$

presents no challenge, but the other term is more interesting. We must differentiate the softmax function by its inputs, then the units by their parameters. More notation: assume we have a vector valued function of vector inputs, for example, $\mathbf{s}(\mathbf{o})$. Here $\mathbf{s}$ is the function and $\mathbf{o}$ are the inputs. I will write $\#(\mathbf{o})$ to mean the number of components of $\mathbf{o}$, and o_i for the i'th component. The matrix of first partial derivatives is extremely important (we will see a lot of these; pay attention). I will write $\mathcal{J}_{\mathbf{s};\mathbf{o}}$ to mean

$$\begin{pmatrix} \frac{\partial s_1}{\partial o_1} & \dots & \frac{\partial s_1}{\partial o_{\#(\mathbf{o})}} \\ \dots & \dots & \dots \\ \frac{\partial s_{\#(\mathbf{s})}}{\partial o_1} & \dots & \frac{\partial s_{\#(\mathbf{s})}}{\partial o_{\#(\mathbf{o})}} \end{pmatrix}$$

and refer to such a matrix of first partial derivatives as a **Jacobian**.

Now we can use the chain rule to write

$$\nabla_{\theta_u} \frac{1}{M} \sum_{i \in \text{minibatch}} \left[\left\{-\mathbf{y}_i^T \log \mathbf{s}(\mathbf{x}_i, \theta)\right\}\right] = \frac{1}{M} \sum_{i \in \text{minibatch}} \left[\left\{-\mathbf{y}_i^T \mathcal{J}_{\log \mathbf{s};\mathbf{o}} \mathcal{J}_{\mathbf{o};\theta_u}\right\}\right].$$

This isn't particularly helpful without knowing the relevant Jacobians. They're quite straightforward.

Write $\mathbb{I}_{[u=v]}(u, v)$ for the indicator function that is 1 when $u = v$ and zero otherwise. We have

$$\begin{aligned} \frac{\partial \log s_u}{\partial o_v} &= \mathbb{I}_{[u=v]} - \frac{e^{o_v}}{\sum_k e^{o_k}} \\ &= \mathbb{I}_{[u=v]} - s_v. \end{aligned}$$

To get the other Jacobian, we need yet more notation (but this isn't new, it's a reminder). I will write $w_{u,i}$ for the i'th component of $\mathbf{w}_u$, and $\mathbb{I}_{[o_u>0]}(o_u)$ for the indicator function that is 1 if its argument is greater than zero. Then

$$\frac{\partial o_u}{\partial w_{u,i}} = x_i \mathbb{I}_{[o_u>0]}(o_u)$$

and

$$\frac{\partial o_u}{\partial b_u} = \mathbb{I}_{[o_u>0]}(o_u).$$

Notice that if $v \neq u$,

$$\frac{\partial o_u}{\partial w_{v,i}} = 0 \text{ and } \frac{\partial o_u}{\partial b_v} = 0.$$

At least in principle, we can build a multiclass classifier in a straightforward way using minibatch gradient descent. We use one unit per class, each one using each component of the feature vector. We obtain training data, and then iterate computing a gradient from a minibatch, and taking a step along the negative of the gradient. If you try, you may run into some of the important small practical problems that cause networks to work badly. Here are some of the ones you may encounter.

Initialization: You need to choose the initial values of all of the parameters. There are many parameters; in our case, with a d dimensional $\mathbf{x}$ and C classes, we have $(d+1) \times C$ parameters. If you initialize each parameter to zero, you will find that the gradient is also zero, which is not helpful. This occurs because all the o_u will be zero (because the $w_{u,i}$ and the b_u are zero). It is usual to initialize to draw a sample of a zero mean normal random variable for each initial value (appropriate choices of variance get interesting; more below).

Learning rate: Each step will look like $\theta^{(n+1)} = \theta^{(n)} - \eta_n \nabla_\theta \text{cost}$. You need to choose η_n for each step. This is widely known as the **learning rate**; an older term is **steplength** (neither term is a super-accurate description). It is not usual for the learning rate to be the same throughout learning. We would like to take "large" steps early, and "small" steps late, in learning, so we would like η_n to be "large" for small n, and "small" for large n. It is tough to be precise about a good choice. As in stochastic gradient descent for a linear SVM, breaking learning into epochs ($e(n)$ is the epoch of the n'th iteration), then choosing two constants a and b to obtain

$$\eta_n = \frac{1}{a + be(n)}$$

is quite a good choice. The constants, and the epoch size, will need to be chosen by experiment. As we build more complex collections of units, the need for a better process will become pressing; two options appear below.

Ensuring learning is proceeding: We need to keep track of what is going on inside the system as we train it. One way is to plot the loss as a function of the number of steps. These plots can be very informative (Figure **??**). If the learning rate is small, the system will make very slow progress but may (eventually) end up in a good state. If the learning rate is large, the system will make fast progress initially, but will then stop improving, because the state will change too quickly to find a good solution. If the learning rate is very large, the system might even diverge. If the learning rate is just right, you should get fast descent to a good value, and then slow but fairly steady improvement. Of course, just as in the case of SVMs, the plot of loss against step isn't a smooth curve, but rather noisy. There is an amusing collection of examples of training problems at lossfunctions.tumblr.com It is quite usual to plot the error rate, or the accuracy, on a validation dataset while training. This will allow you to compare the training error with the validation error. If these are very different, you have a problem: the system is overfitting, or not generalizing well. You should increase the regularization constant.

Dead units: Imagine the system gets into a state where for some unit u, $o_u = 0$ for every training data item. This could happen, for example, if the learning rate was too large. Then it can't get out of this state, because the gradient for that unit will be zero for every training data item, too. Such units are referred to as **dead units**. This problem can be contained by keeping the learning rate small enough. In more complex architectures (below), it is also contained by having a large number of units.

Gradient problems: There are a variety of important ways to have gradient problems. By far the most important is making a simple error in code (i.e you compute the Jacobian elements wrong). This is surprisingly common; everybody does it at least once; and one learns to check gradients. Checking is fairly straightforward. You compute a numerical derivative, and compare that to the exact derivative. If they're too different, you have a gradient problem you need to fix. We will see a second important gradient problem when we see more complex architectures.

Choosing the regularization constant: This follows the recipe we saw for a linear SVM. Hold out a validation dataset. Train for several different values of λ. Evaluate each system on the validation dataset, and choose the best. Notice this involves many rounds of training, which could make things slow.

Does it work? Evaluating the classifier we have described is like evaluating any other classifier. You evaluate the error on a held-out data set that *wasn't* used to choose the regularization constant, or during training.

12.2 LAYERS AND NETWORKS

We have built a multiclass classifier out of units by using one unit per class, then interpreting the outputs of the units as probabilities using a softmax function. This classifier is at best only mildly interesting. The way to get something really interesting is to ask what the features for this classifier should be. To date, we have not looked closely at features. Instead, we've assumed they "come with the dataset" or should be constructed from domain knowledge. Remember that, in the case of regression, we could improve predictions by forming non-linear functions of features. We can do better than that; we could *learn* what non-linear functions to apply, by using the output of one set of units to form the inputs of the next set.

We will focus on systems built by organizing the units into **layers**; these layers form a **neural network** (a term I dislike, for the reasons above, but use because everybody else does). There is an input layer, consisting of the units that receive feature inputs from outside the network. There is an output layer, consisting of units whose outputs are passed outside the network. These two might be the same, as they were in the previous section. The most interesting cases occur when they are not the same. There may be **hidden layers**, whose inputs come from other layers and whose outputs go to other layers. In our case, the layers are ordered, and outputs of a given layer act as inputs to the next layer only (as in Figure 12.2 - we don't allow connections to wander all over the network). For the moment, assume that each unit in a layer receives an input from every unit in the previous layer;

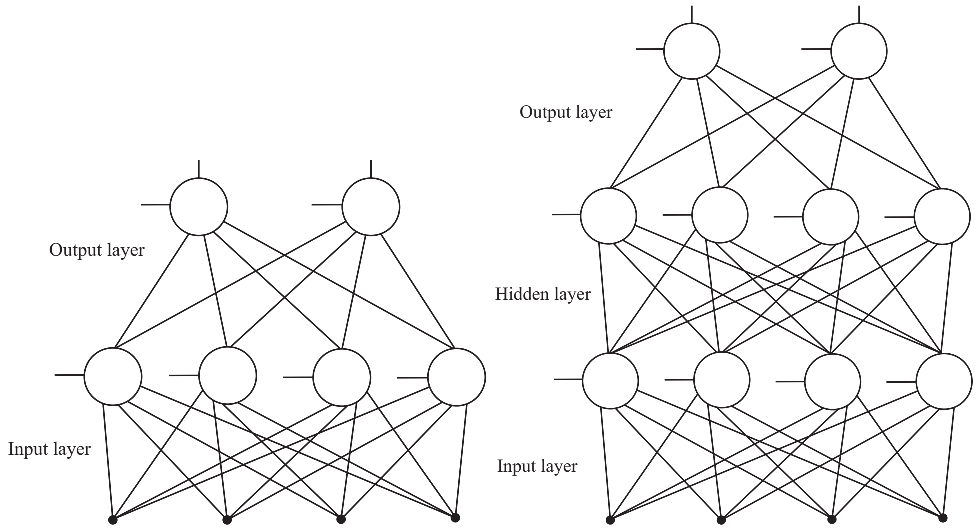


FIGURE 12.2: *On the* **left***, an input layer connected to an output layer. The units in the input layer take the inputs and compute features; the output layer turns these features into output values that will be turned into class probabilities with the softmax function. On the* **right***, there is a hidden layer between input and output layer. This architecture means that the features seen by the output layer can be trained to be a significantly more complex function of the inputs.*

this means that our network is **fully connected**. Other architectures are possible, but right now the most important question is how to train the resulting object.

12.2.1 Notation

Inevitably, we need yet more notation. There will be L layers. The input layer is layer 1, and the output layer is L. I will write u_i^l for the i'th unit in the l'th layer. This unit has output o_i^l and parameters $\mathbf{w}_i^l$ and b_i^l, which I will stack into a vector θ_i^l. I write θ^l to refer to all the parameters of layer l. If I do not need to identify the layer in which a unit sits (for example, if I am summing over all units) I will drop the superscript. The vector of inputs to this unit is $\mathbf{x}_i^l$. These inputs are formed by choosing from the outputs of layer $l-1$. I will write $\mathbf{o}^l$ for all the outputs of the l'th layer, stacked into a vector. I will represent the connections by a matrix $\mathcal{C}_i^l$, so that $\mathbf{x}_i^l = \mathcal{C}_i^l \mathbf{o}^{l-1}$. The matrix $\mathcal{C}_i^l$ contains only 1 or 0 entries, and in the case of fully connected layers, it is the identity. Notice that every unit has its own $\mathcal{C}_i^l$.

I will write $L(\mathbf{y}_i, \mathbf{s}(\mathbf{o}^L(\mathbf{x}_i, \theta)))$ for the loss of classifying the i'th example using softmax. We will continue to use

$$L(\mathbf{y}_i, \mathbf{s}(\mathbf{o}^L(\mathbf{x}_i, \theta))) = -\mathbf{y}_i^T \log \mathbf{s}(\mathbf{o}^L(\mathbf{x}_i, \theta)))$$

but in other applications, other losses might arise.

Generally, we will train by mini batch gradient descent, though I will describe some tricks that can speed up training and improve results. But we must compute

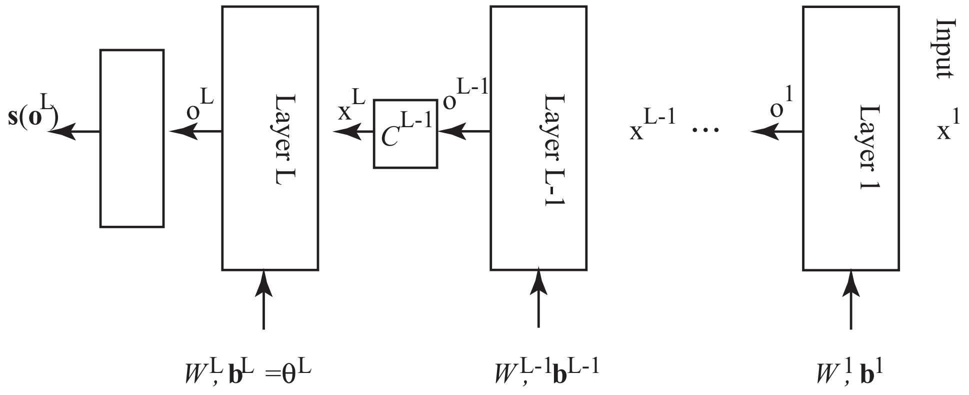


FIGURE 12.3: *Notation for layers, inputs, etc.*

the gradient. The output layer of our network has C units, one per class. We will apply the softmax to these outputs, as before. Writing E for the cost of error on training examples and R for the regularization term, we can write the cost of using the network as

$$\text{cost} = E + R = (1/N) \sum_{i \in \text{examples}} L(\mathbf{y}_i, \mathbf{s}(\mathbf{o}^L(\mathbf{x}_i, \theta))) + \frac{\lambda}{2} \sum_{k \in \text{units}} \mathbf{w}_k^T \mathbf{w}_k.$$

You should not let this compactified notation let you lose track of the fact that $\mathbf{o}^L$ depends on $\mathbf{x}_i$ through $\mathbf{o}^{L-1}, \ldots, \mathbf{o}^1$. What we really should write is

$$\mathbf{o}^L(\mathbf{o}^{L-1}(\ldots(\mathbf{o}^1(\mathbf{x}, \theta^1), \theta^2), \ldots), \theta^L).$$

Equivalently, we could stack all the $\mathcal{C}_i^l$ into one linear operator $\mathcal{C}^l$ and write

$$\begin{array}{rcl}
\mathbf{o}^L(\mathbf{x}^L, \theta^L) & & \text{where} \\
\mathbf{x}^L & = & \mathcal{C}^L \mathbf{o}^{L-1}(\mathbf{x}^{L-1}, \theta^{L-1}) \\
\ldots & = & \ldots \\
\mathbf{x}^2 & = & \mathcal{C}^2 \mathbf{o}^1(\mathbf{x}^1, \theta^1) \\
\mathbf{x}^1 = \mathcal{C}^1 \mathbf{x} & &
\end{array}$$

This is important, because it allows us to write an expression for the gradient.

12.2.2 Training, Gradients and Backpropagation

Now consider $\nabla_\theta E$. We have that E is a sum over examples. The gradient of the loss at a particular example is of most interest, because we will usually train with minibatches. So we are interested in

$$\nabla_\theta E_i = \nabla_\theta L(\mathbf{y}_i, \mathbf{s}(\mathbf{o}^L(\mathbf{x}_i, \theta))) = \nabla_\theta \left[-\mathbf{y}_i^T \log \mathbf{s}(\mathbf{o}^L(\mathbf{x}_i, \theta)) \right]$$

and we can extend our use of the chain rule from section 12.1.3, very aggressively. We have

$$\nabla_\theta^L L(\mathbf{y}_i, \mathbf{s}(\mathbf{o}^L(\mathbf{x}_i, \theta))) = -\mathbf{y}_i^T \mathcal{J}_{\log \mathbf{s}; \mathbf{o}^L} J_{\mathbf{o}^L; \theta^L}$$

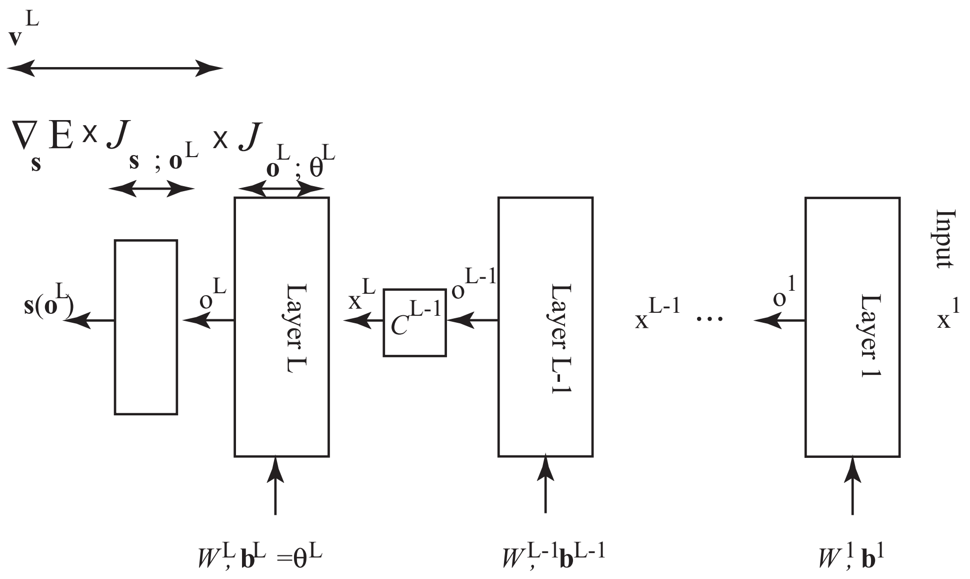


FIGURE 12.4: *Constructing the gradient with respect to θ^L.*

as in that section. Differentiating with respect to θ^{L-1} is more interesting. Layer L depends on θ^{L-1} in a somewhat roundabout way; layer $L-1$ uses θ^{L-1} to produce its outputs, and these are fed into layer L *as its inputs*. So we must have

$$\nabla_\theta^{L-1} E_i = -\mathbf{y}_i^T \mathcal{J}_{\log \mathbf{s};\mathbf{o}^L} J_{\mathbf{o}^L;\mathbf{x}^L} J_{\mathbf{x}^L;\theta^{L-1}}$$

(look carefully at the subscripts on the Jacobians). These Jacobians have about the same form as those in section 12.1.3 if you recall that $\mathbf{x}^L = \mathcal{C}^L \mathbf{o}^{L-1}$. In turn, this means that

$$J_{\mathbf{x}^L;\theta^{L-1}} = \mathcal{C}^L J_{\mathbf{o}^{L-1};\theta^{L-1}}$$

and the form of that Jacobian appears in section 12.1.3. But $\mathbf{o}^L$ depends on θ^{L-2} through *its* inputs (which are $\mathbf{x}^{L-1}$), so that

$$\nabla_\theta^{L-2} E_i = -\mathbf{y}_i^T \mathcal{J}_{\log \mathbf{s};\mathbf{o}^L} J_{\mathbf{o}^L;\mathbf{x}^L} J_{\mathbf{o}^{L-1};\mathbf{x}^{L-1}} J_{\mathbf{x}^{L-1};\theta^{L-2}}$$

(again, look carefully at the subscripts on each of the Jacobians).

We can now get to the point. We have a recursion, which can be made more

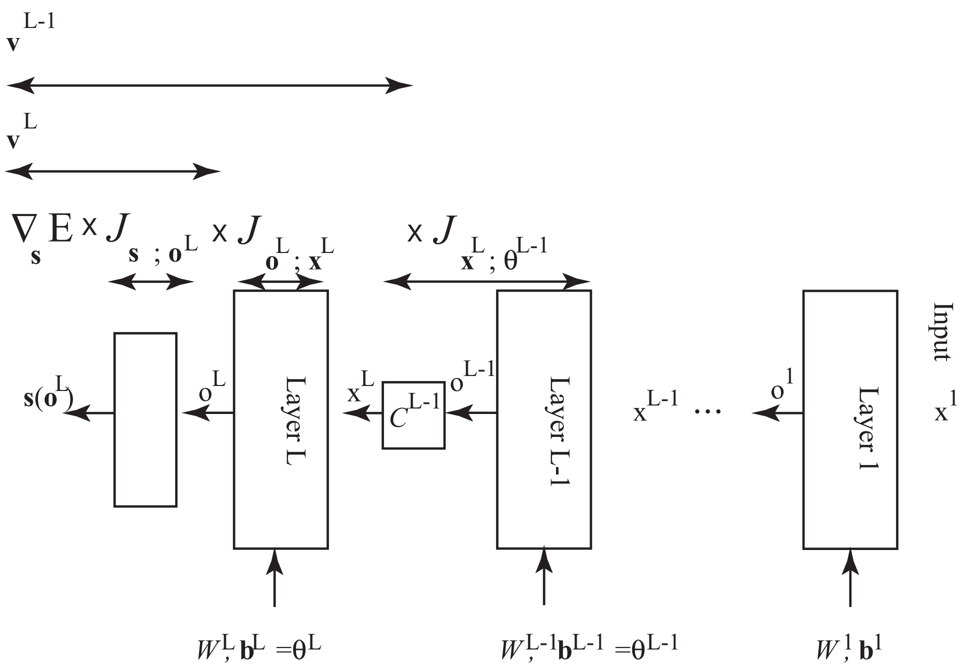


FIGURE 12.5: *Constructing the gradient with respect to* θ^{L-1}.

obvious with some notation. We have

$$
\begin{array}{rcl}
\mathbf{v}^L & = & \left(\nabla_{\mathbf{o}}^L E_i\right) \\
\nabla_{\theta^L} E_i & = & \mathbf{v}^L \mathcal{J}_{\mathbf{o}^L;\theta^L} \\
\mathbf{v}^{L-1} & = & \mathbf{v}^L \mathcal{J}_{\mathbf{o}^L;\mathbf{x}^L} \\
\nabla_{\theta^{L-1}} E & = & \mathbf{v}^{L-1} \mathcal{J}_{\mathbf{x}^L;\theta^{L-1}} \\
\ldots & & \\
\mathbf{v}^{i-1} & = & \mathbf{v}^i \mathcal{J}_{\mathbf{x}^{i+1};\mathbf{x}^i} \\
\nabla_{\theta^{i-1}} E & = & \mathbf{v}^{i-1} \mathcal{J}_{\mathbf{x}^i;\theta^{i-1}} \\
\ldots & &
\end{array}
$$

I have not added notation to keep track of the point at which the partial derivative is evaluated (it should be obvious, and we have quite enough notation already). When you look at this recursion, you should see that, to evaluate $\mathbf{v}^{i-1}$, you will need to know $\mathbf{x}^k$ for $k \geq i-1$. This suggests the following strategy. We compute the $\mathbf{x}$'s (and, equivalently, $\mathbf{o}$'s) with a "forward pass", moving from the input layer to the output layer. Then, in a "backward pass" from the output to the input, we compute the gradient. Doing this is often referred to as **backpropagation**.

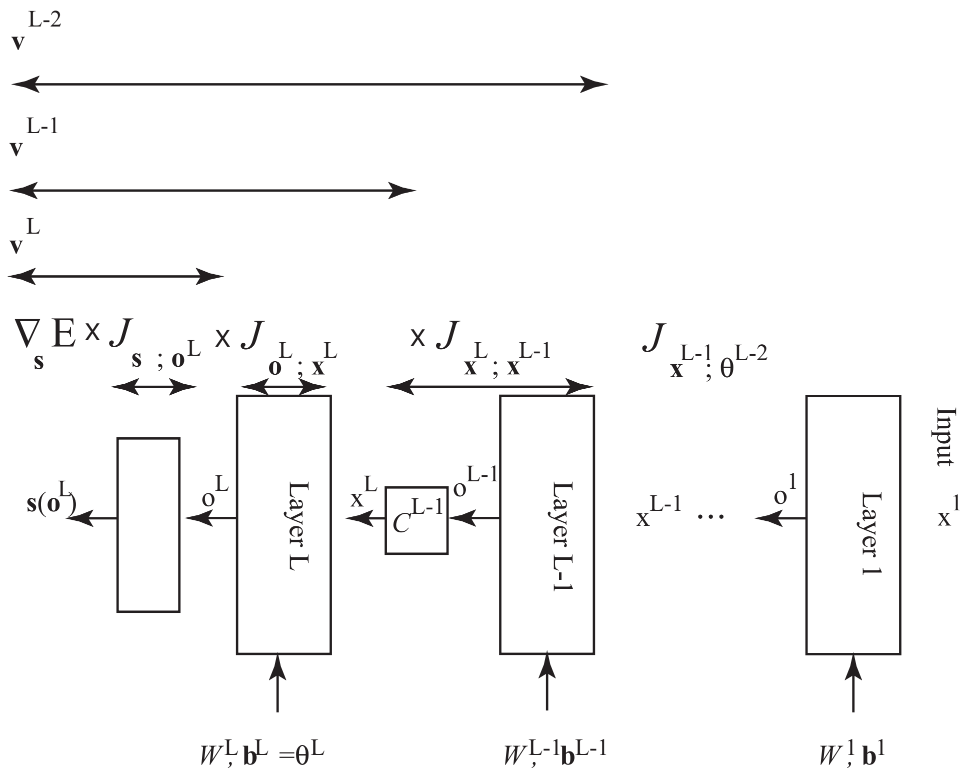


FIGURE 12.6: *Constructing the gradient with respect to* θ^{L-2}.

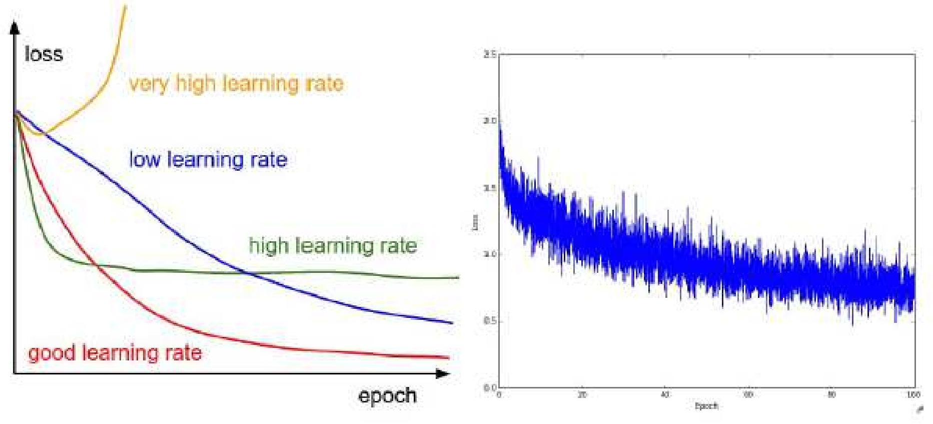


FIGURE 12.7: *On the* **left**, *a cartoon of the error rate encountered during training a multilayer network and the main phenomena you might observe. On the* **right**, *an actual example. Each of these figures is from the excellent course notes for the Stanford class cs231n Convolutional Neural Networks for Visual Recognition, written by Andrej Karpathy. You can find these notes at:* http://cs231n.stanford.edu.

12.2.3 Training Multiple Layers

A multilayer network represents an extremely complex, highly non-linear function, with an immense number of parameters. Training such networks is not easy. Neural networks are quite an old idea, but have only relatively recently had impact in practical applications. Hindsight suggests the problem is that networks are hard to train successfully. There is now a collection of quite successful tricks — I'll try to describe the most important — but the situation is still not completely clear.

The simplest training strategy is minibatch gradient descent. At round r, we have the set of weights $\theta^{(r)}$. We form the gradient for a minibatch $\nabla_\theta E$, and update the weights by taking a small step $\eta^{(r)}$ (usually referred to as the **learning rate**) backwards along the gradient, yielding

$$\theta^{(r+1)} = \theta^{(r)} - \eta^{(r)} \nabla_\theta E.$$

The most immediate difficulties are where to start, and what is $\eta^{(r)}$.

Initialization: As for a single layer of units, it is a bad idea to initialize each parameter to zero. It is usual to draw a sample of a zero mean normal random variable for each initial value. However, in a multilayer network, we may well have some units receiving input from more (or fewer) units than others (this is referred to as the **fan in** of the unit). Now assume that we have two units: one with many inputs, and one with few. If we initialize each units weights using the same zero mean normal random variable, the unit with more inputs will have a higher variance output (I'm ignoring the nonlinearity). This tends to lead to problems, because units at the next level will see unbalanced inputs. Experiment has shown that it is a good idea to allow the variance of the random variable you sample to depend on the fan in of the unit whose parameter you are initializing. Write n for the fan in of the unit in question, and ϵ for a small non-negative number. Current best practice appears to be that one initializes each weight with an independent sample of a random variable with mean 0 and *variance*

$$\epsilon \frac{\sqrt{2}}{n}.$$

Choosing ϵ too small or too big can lead to trouble, but I'm not aware of any recipe for coming up with a good choice. Typically, biases are initialized either to 0, or to a small non-negative number; there is mild evidence that 0 is a better choice.

Learning rate: The remarks above about learning rate apply, but for more complicated networks it is usual to apply one of the methods of section 12.2.4, which adjust the gradient to get better optimization behavior.

Ensuring learning is proceeding: We need to keep track of what is going on inside the system as we train it. One way is to plot the loss as a function of the number of steps. These plots can be very informative (Figure 12.7). If the learning rate is small, the system will make very slow progress but may (eventually) end up in a good state. If the learning rate is large, the system will make fast progress initially, but will then stop improving, because the state will change too quickly to find a good solution. If the learning rate is very large, the system might even diverge. If the learning rate is just right, you should get fast descent to a good value, and then slow but fairly steady improvement. Of course, just as in the case of SVMs, the plot of loss against step isn't a smooth curve, but rather noisy. There is an amusing collection of examples of training problems at lossfunctions.tumblr.com It is quite usual to plot the error rate, or the accuracy, on a validation dataset while training. This will allow you to compare the training error with the validation error. If these are very different, you have a problem: the system is overfitting, or not generalizing well. You should increase the regularization constant.

Dead units: The remarks above apply.

Gradient problems: The remarks above apply.

Choosing the regularization constant: The remarks above apply, but for more complex networks, it is usual to use the more sophisticated regularization described in section **??** (at considerable training cost).

Does it work? Evaluating the classifier we have described is like evaluating any other classifier. You evaluate the error on a held-out data set that *wasn't* used to choose the regularization constant, or during training.

12.2.4 Gradient Scaling Tricks

Everyone is surprised the first time they learn that the best direction to travel in when you want to minimize a function is not, in fact, backwards down the gradient. The gradient *is* uphill, but repeated downhill steps are often not particularly efficient. An example can help, and we will look at this point several ways because different people have different ways of understanding this point.

We can look at the problem with algebra. Consider $f(x, y) = (1/2)(\epsilon x^2 + y^2)$, where ϵ is a small positive number. The gradient at (x, y) is $[\epsilon x, y]$. For simplicity, use a fixed learning rate η, so we have $\left[x^{(r)}, y^{(r)}\right] = \left[(1-\epsilon\eta)x^{(r-1)}, (1-\eta)y^{(r-1)}\right]$. If you start at, say, $(x^{(0)}, y^{(0)})$ and repeatedly go downhill along the gradient, you will travel very slowly to your destination. You can show that $\left[x^{(r)}, y^{(r)}\right] = \left[(1-\epsilon\eta)^r x^{(0)}, (1-\eta)^r y^{(0)}\right]$. The problem is that the gradient in y is quite large (so y must change quickly) and the gradient in x is small (so x changes slowly). In turn, for steps in y to converge we must have $|1-\eta| < 1$; but for steps in x to converge, we require only the much weaker constraint $|1-\epsilon\eta| < 1$. Imagine we choose the largest η we dare for the y constraint. The y value will very quickly have small magnitude, though its sign will change with each step. But the x steps will move you closer to the right spot only extremely slowly.

Another way to see this problem is to reason geometrically. Figure 12.8 shows this effect for this function. The gradient is at right angles to the level curves of the function. But when the level curves form a narrow valley, the gradient points across the valley rather than down it. The effect isn't changed by rotating and translating the function (Figure 12.9).

You may have learned that Newton's method resolves this problem. This is all very well, but to apply Newton's method we would need to know the matrix of second partial derivatives. A network can easily have thousands to millions of parameters, and we simply can't form, store, or work with matrices of these dimensions. Instead, we will need to think more qualitatively about what is causing trouble.

One useful insight into the problem is that fast changes in the gradient vector are worrying. For example, consider $f(x) = (1/2)(x^2 + y^2)$. Imagine you start

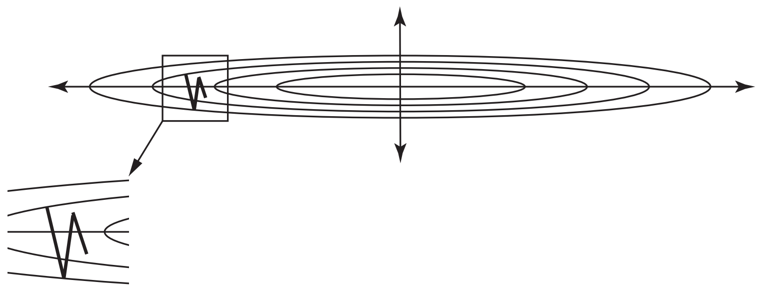


FIGURE 12.8: *A plot of the level curves (curves of constant value) of the function $f(x, y) = (1/2)(\epsilon x^2 + y^2)$. Notice that the value changes slowly with large changes in x, and quickly with small changes in y. The gradient points mostly toward the x-axis; this means that gradient descent is a slow zig-zag across the "valley" of the function, as illustrated. We might be able to fix this problem by changing coordinates,* if *we knew what change of coordinates to use.*

far away from the origin. The gradient won't change much along reasonably sized steps. But now imagine yourself on one side of a valley like the function $f(x) = (1/2)(x^2 + \epsilon y^2)$; as you move along the gradient, the gradient in the x direction gets smaller very quickly, then points back in the direction you came from. You are not justified in taking a large step in this direction, because if you do you will end up at a point with a very different gradient. Similarly, the gradient in the y direction is small, and stays small for quite large changes in y value. You would like to take a small step in the x direction and a large step in the y direction.

You can see that this is the impact of the second derivative of the function (which is what Newton's method is all about). But we can't do Newton's method. We would like to travel further in directions where the gradient doesn't change much, and less far in directions where it changes a lot. There are several methods for doing so.

Momentum: We should like to discourage parameters from "zig-zagging" as in the example above. In these examples, the problem is caused by components of the gradient changing sign from step to step. It is natural to try and smooth the

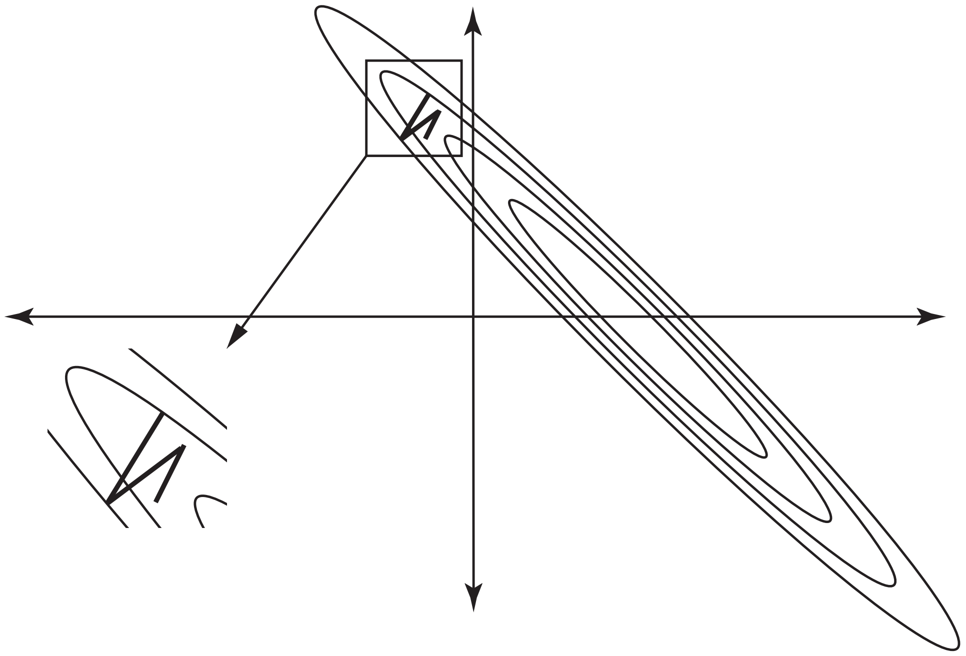


FIGURE 12.9: *Rotating and translating a function rotates and translates the gradient; this is a picture of the function of figure 12.8, but now rotated and translated. The problem of zig-zagging remains. This is important, because it means that we may have serious difficulty choosing a good change of coordinates.*

gradient. We could do so by forming a moving average of the gradient. Construct a vector $\mathbf{v}$, the same size as the gradient, and initialize this to zero. Choose a positive number $\mu < 1$. Then we iterate

$$\begin{aligned}\mathbf{v}^{(r+1)} &= \mu * \mathbf{v}^{(r)} + \eta \nabla_\theta E \\ \theta^{(r+1)} &= \theta^{(r)} - \mathbf{v}^{(r+1)}\end{aligned}$$

Notice that, in this case, the update is an average of all past gradients, each weighted by a power of μ. If μ is small, then only relatively recent gradients will participate in the average, and there will be less smoothing. Larger μ lead to more smoothing. A typical value is $\mu = 0.9$. It is reasonable to make the learning rate go down with epoch when you use momentum, but keep in mind that a very large μ will mean you need to take several steps before the effect of a change in learning rate shows.

Adagrad: We will keep track of the size of each component of the gradient. In particular, we have a running cache $\mathbf{c}$ which is initialized at zero. We choose a small number α (typically 1e-6), and a fixed η. Write $g_i^{(r)}$ for the i'th component of the gradient $\nabla_\theta E$ computed at the r'th iteration.Then we iterate

$$\begin{aligned} c_i^{(r+1)} &= c_i^{(r)} + (g_i^{(r)})^2 \\ \theta_i^{(r+1)} &= \theta_i^{(r)} - \eta \frac{g_i^{(r)}}{(c_i^{(r+1)})^{\frac{1}{2}} + \alpha} \end{aligned}$$

Notice that each component of the gradient has its own learning rate, set by the history of previous gradients.

RMSprop: This is a modification of Adagrad, to allow it to "forget" large gradients that occurred far in the past. We choose another number, Δ, (the **decay rate**; typical values might be 0.9, 0.99 or 0.999), and iterate

$$\begin{aligned} c_i^{(r+1)} &= \Delta c_i^{(r)} + (1-\Delta)(g_i^{(r)})^2 \\ \theta_i^{(r+1)} &= \theta_i^{(r)} - \eta \frac{g_i^{(r)}}{(c_i^{(r+1)})^{\frac{1}{2}} + \alpha} \end{aligned}$$

12.2.5 Dropout

Regularizing by the square of the weights is all very well, but quite quickly we will have problems because there are so many weights. An alternative, and very useful, regularization strategy is to try and ensure that no unit relies too much on the output of any other unit. One can do this as follows. At each training step, randomly select some units, set their outputs to zero (and reweight the inputs of the units receiving input from them), and then take the step. Now units are trained to produce reasonable outputs even if some of their inputs are randomly set to zero — units can't rely too much on one input, because it might be turned off. Notice that this sounds sensible, but it isn't quite a proof that the approach is sound; that comes from experiment. The approach is known as **dropout**.

There are some important details we can't go into. Output units are not subject to dropout, but one can also turn off inputs randomly. At test time, there is no dropout. Every unit computes its usual output in the usual way. This creates an important training issue. Write p for the probability that a unit is dropped out, which will be the same for all units subject to dropout. You should think of the expected output of the i'th unit at *training* time as $(1-p)o_i$ (because with probability p, it is zero). But at test time, the next unit will see o_i; so at training time, you should reweight the inputs by $1/(1-p)$. In exercises, we will use packages that arrange all the details for us.

12.2.6 It's Still Difficult..

All the tricks above are helpful, but training a multilayer neural network is still difficult. Fully connected layers have many parameters. It's quite natural to take an input feature vector of moderate dimension, build one layer that produces a much higher dimensional vector, then stack a series of quite high dimensional layers on top of that. There is quite good evidence that having many layers can improve practical performance *if* one can train the resulting network. Such an architecture has been known for a long time, but hasn't been particularly successful until recently.

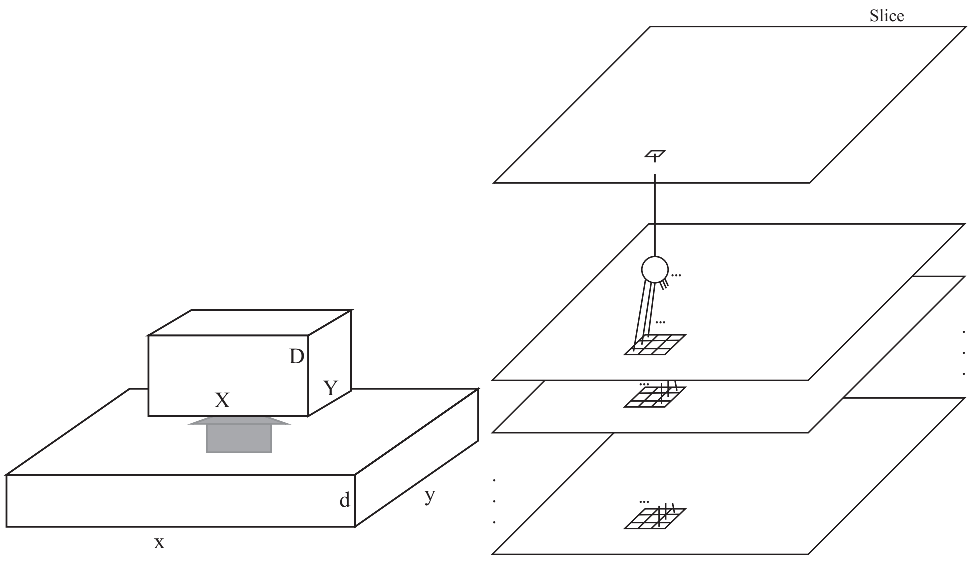


FIGURE 12.10: *Terminology for building a convolutional layer. On the* **left***, a layer turns one block of data into another. The x and y coordinates index the position of a location in the block, and the d coordinate identifies the data item at that point. A natural example of a block is a color image, which usually has three layers (d=3); then x and y identify the pixel, and d chooses the R, G, or B slice. The dimensions x and y are the* spatial *dimensions. On the right* **right***, a unit in a convolutional layer forms a weighted sum of set of locations, adds a bias, then applies a RELU. There is one unit for each (X, Y, D) location in the output block. For each (X, Y) in the output block, there is a corresponding window of size* (w_x, w_y) *in the (x, y) space. Each of the D units whose responses form the D values at the (X, Y) location in the output block forms a weighted sum of all the values covered by that window. These inputs come from each slice in the window below that unit (so the number of inputs is* $d \times w_x \times w_y$*). Each of the units that feed the a particular slice in the output block has the same set of weights, so you should think of a unit as a form of pattern detector; it will respond strongly if the block below it is "similar" to the weights.*

There are several structural obstacles. Without GPU's, evaluating such a network can be slow, making training slow. The number of parameters in just one fully connected layer is high, meaning that multiple layers will need a lot of data to train, and will take many training batches. There is some reason to believe that multilayer neural networks were discounted in application areas for quite a long time because people underestimated just how much data and how much training was required to make them perform well.

One obstacle that remains technically important has to do with the gradient. Look at the recursion I described for backpropagation. The gradient update at the L'th (top) layer depends pretty directly on the parameters in that layer. But now consider a layer close to the input end of the network. The gradient update has

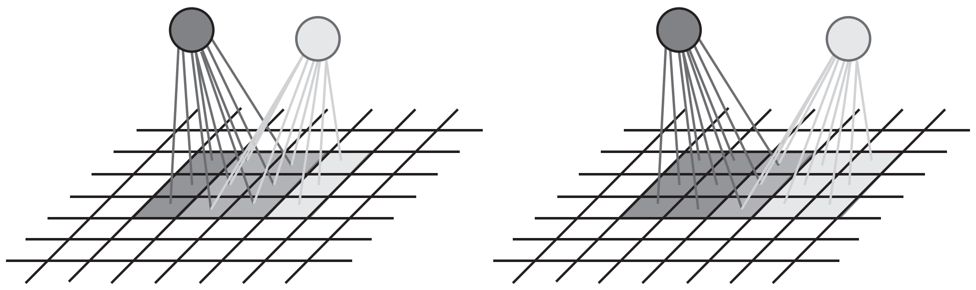


FIGURE 12.11: *Figure 12.10 shows units taking an input block and creating an output block. Each unit is fed by a window on the spatial dimensions of the input block. The window is advanced by the* stride *to feed the next unit. On the* **left***, two units fed by 3x3 windows with a stride of 1. I have shaded the pixels in the window to show which pixels go to which unit; notice the units share 6 pixels. In this case, the spatial dimensions of the output block will be either the same as those of the input block (if we find some values to feed pixels in windows that hang over the edge of the image), or only slightly smaller (if we ignore units whose windows hang over the edges of the image). On the* **right***, two units fed by 3x3 windows with a stride of 2. Notice the units share fewer pixels, and the output block will be smaller than the input block.*

been multiplied by several Jacobian matrices. The update may be very small (if these Jacobians shrink their input vectors) or unhelpful (if layers close to the output have poor parameter estimates). For the gradient update to be really helpful, we'd like the layers higher up the network to be right; but we can't achieve this with lower layers that are confused, because they pass their outputs up. If a layer low in the network is in a nonsensical state, it may be very hard to get it out of that state. In turn, this means that adding layers to a network might improve performance, but also might make it worse because the training turns out poorly.

There are a variety of strategies for dealing with this problem. We might just train for a very long time, possibly using gradient rescaling tricks. We might reduce the number of parameters in the layers, by passing to convolutional layers (below) rather than fully connected layers. We might use various tricks to initialize each layer with a good estimate. This is a topic of widespread current interest, but one I can't deal with in any detail here. Finally, we might use architectural tricks (section 10.1) to allow inputs to bypass layers, so that poorly trained layers create fewer difficulties.

12.3 CONVOLUTIONAL NEURAL NETWORKS

One area where neural networks have had tremendous impact is in image understanding. Images have special properties that motivate special constructions of features. These constructions yield a layer architecture that has a significantly reduced number of parameters in the layer. This architecture has proven useful in other applications, but we will work with images.

12.3.1 Images and Convolutional Layers

Using the output of one layer to form features for another layer is attractive. The natural consequence of this idea is that the input to the network would be the image pixels. But this presents some important difficulties. There are an awful lot of pixels in most images, and that means it's likely that there will be an awful lot of parameters.

For a variety of reasons, it doesn't make much sense to have an input layer consisting of units each of which sees all the image pixels. There would be a tremendous number of weights to train. It would be hard to explain why units have different values for the weights. Instead, we can use some intuitions from computer vision.

First, we need to build systems that can handle different versions of what is essentially the same image. For example, imagine you turn the camera slightly to one side, or raise it or lower it when taking the image. If every input unit sees every pixel, this would constrain the form of the weights. Turning the camera up a bit shifts the image down a bit; the representation shouldn't be severely disrupted by this. Second, long experience in computer vision has produced a (very rough) recipe for building image features: you construct features that respond to patterns in small, localized neighborhoods; then other features look at patterns of *those* features; then others look at patterns of those, and so on (*big fleas have little fleas upon their backs to bite 'em; and little fleas have smaller ones, and so ad infinitum*).

We will assume that images are 3D. The first two dimensions will be the x and y dimensions in the image, the third (for the moment!) will identify the color layer of the image (for example, R, G and B). We will build layers that take 3D objects like images (which I will call **blocks**) and make new blocks (Figure 12.10; notice the input block has dimension $x \times y \times d$ and the output block has dimension $X \times Y \times D$). Each block is a stack of **slices**, which — like color layers in an image — have two spatial dimensions.

These layers will draw from a standard recipe for building an image feature that describes a small neighborhood. We construct a **convolution kernel**, which is a small block. This is typically odd sized, and typically from 3×3 to a few tens by a few tens in size along the spatial dimensions, and is always of size d in the other dimension.

Write $\mathcal{I}$ for a block (for example, an image), $\mathcal{K}$ for the kernel, b for a bias term (which might be zero; some, but not all, convolutional layers use a bias term), and $\mathcal{I}_{ijk}$ for the i, j'th pixel in the k'th slice of the block. Write F for the function implemented by a RELU, so that $F(x) = \max(0, x)$. Now we form a slice $\mathcal{O}$, whose u, v'th entry is

$$\mathcal{O}_{uv} = F(\mathcal{W}_{uv} + b) = F(\sum_{ijk} \mathcal{I}_{u+i,v+j,k}\mathcal{K}_{ijk} + b),$$

where I am assuming the sum goes over all values of i and j, and if the indices to either $\mathcal{I}$ or $\mathcal{K}$ go outside the domain, then the reported value is zero. There is room for some confusion here, because one can use a variety of different indexing schemes, and different authors use different ones (usually for compatibility with the history of convolution); this is of no significance. Figure 12.10 illustrates the process that produces a slice from a block.

The operation that produces $\mathcal{W}$ from $\mathcal{I}$ and $\mathcal{K}$ is known as **convolution**, and it is usual to write $\mathcal{W} = \mathcal{K} * \mathcal{I}$. We will not go into all the properties of convolution, but you should notice one extremely important property. We obtain the value at a pixel by centering $\mathcal{K}$ on that pixel. We now have a patch sitting over all the layers of the image at some location; we multiply the pixels in that patch (by layer) by the corresponding image pixels (in layers), then accumulate the products — the result goes into $\mathcal{O}$. This is like a dot-product — we will get a large positive value in $\mathcal{O}$ at that pixel if the image window around that pixel looks like $\mathcal{K}$, and a small negative value if they're the same up to a sign change. You should think of a convolution kernel as being an example pattern.

Now when you convolve a kernel with an image, you get, at each location, an estimate of how much that image looks like that kernel at that point. The output is the response of a collection of simple pattern detectors, one at each pixel, *for the same pattern*. We may not need every such output; instead, we might look at every second (third, etc.) pixel in each direction. This choice is known as the **stride**. A stride of 1 corresponds to looking at every pixel; of 2, every second pixel; and so on (Figure 12.11)

A slice can be interpreted as a map, giving the response of a local feature detector at every (resp. every second; every third; etc.) pixel. At each pixel of interest (i.e. every pixel; every second, etc. depending on stride), we place a window (which should be odd-sized, to make indexing easier). Every pixel in that window is an input to a unit that corresponds to the window, which multiplies each pixel by a weight, sums all these terms, then applies a RELU. What makes a slice special is that *each unit uses the same set of weights*. The size of this object depends a little on the software package you are using. Assume the input image is of size $n_x \times n_y \times n_z$, and the kernel is of size $2k_x + 1 \times 2k_y + 1 \times n_z$. At least in principle, you cannot place a unit over a pixel that is too close to the edge, because then some of its inputs are outside the image. You could pad the image (either with constants, or by reflecting it, or by attaching copies of the columns/rows at the edge) and supply these inputs; in this case, the output could be $n_x \times n_y \times n_z$. Otherwise, you could place units only over pixels where all of the unit's inputs are inside the image. Then you would have an output of size $n_x - 2k_x \times n_y - 2n_y \times n_z$. The kernel is usually small, so the difference in sizes isn't that great. Most software packages are willing to set up either case.

A slice finds locations in the image where a particular pattern (identified by the weights) occurs. We could attach many slices to the image. They should all have the same stride, so they're all the same size. The output of this collection of slices would be one vector at each pixel location, where the components of the vector represent the similarity between the image patch centered at that location and a particular pattern. A collection of slices is usually referred to as a **convolutional layer**. You should think of a convolutional layer as being like a color image. There are now may different color layers (the slices), so that dimension has been expanded.

12.3.2 Convolutional Layers upon Convolutional Layers

Now the output of the initial convolutional layer is a set of slices, registered to the input image, forming a block of data. That looks like the input of that layer (a set of slices — color layers) forming a **block** of data. This suggests we could use the output of the first convolutional layer could be connected to a second convolutional layer, a second to a third, and so on. Doing so turns out to be an excellent idea.

Think about the output of the first convolutional layer. Each location receives inputs from pixels in a window about that location. Now if we put a second layer on top of the first, each location in the second receives inputs from first layer values in a window about that location. This means that locations in the second layer are affected by a larger window of pixels than those in the first layer. You should think of these as representing “patterns of patterns”. If we place a third layer on top of the second layer, locations in that third layer will depend on an even larger window of pixels. A fourth layer will depend on a yet larger window, and so on.

12.3.3 Pooling

If you have several convolutional layers with stride 1, then each block of data has the same spatial dimensions. This tends to be a problem, because the pixels that feed a unit in the top layer will tend to have a large overlap with the pixels that feed the unit next to it. In turn, the values that the units take will be similar, and so there will be redundant information in the output block. It is usual to try and deal with this by making blocks get smaller. One natural strategy is to occasionally have a layer that has stride 2.

An alternative strategy is to use **max pooling**. A pooling unit reports the largest value of its inputs. In the most usual arrangement, a pooling layer will take an (x, y, d) block to a $(x/2, y/2, d)$ block. For the moment, ignore the entirely minor problems presented by a fractional dimension. The new block is obtained by pooling units that pool a 2x2 window at each slice of the input block to form each slice of the output block. These units are placed so they don’t overlap, so the output block is half the size of the input block (for some reason, this configuration is hard to say but easy to see; Figure 12.12). If x or y or both are odd, there are two options; one could ignore the odd pixel on the boundary, or one could build a row (column; both) of imputed values, most likely by copying the row (column; both) on the edge. These two strategies yield, respectively, floor$(x/2)$ and ceil$(x/2)$ for the new dimension. Pooling seems to be falling out of favor, but not so much or so fast that you will not encounter it.

12.4 BUILDING AN IMAGE CLASSIFIER

There are two problems that lie at the core of image understanding. The first is **image classification**, where we decide what class an image of a fixed size belongs to. The taxonomy of classes is provided in advance, but it’s usual to work with a collection of images of objects. These objects will be largely centered in the image, and largely isolated. Each image will have an associated object name. There are many collections with this structure. The best known, by far, is `ImageNet`, which can be found at http://www.image-net.org. There is a regular competition to

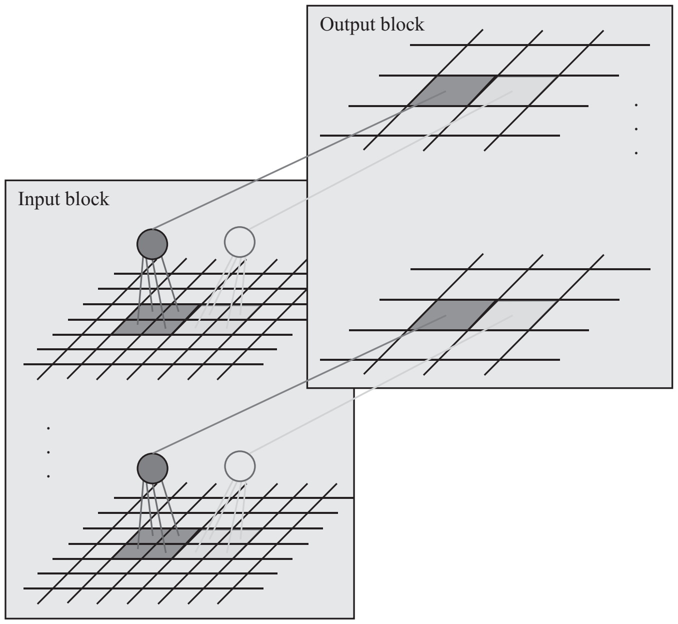


FIGURE 12.12: *In a pooling layer, pooling units compute the largest value of their inputs, then pass it on. The most common case is 2x2, illustrated here. We tile each slice with 2x2 windows that do not overlap. Pooling units compute the max, then pass that on to the corresponding location in the corresponding slice of the output block. As a result, the spatial dimensions of the output block will be about half those of the input block (details depend on how one handles windows that hang over the edge.*

classify ImageNet images. Be aware that, while this chapter tries to give a concise description of best practice, you might need to do more than read it to do well in the competition.

The second problem is **object detection**, where we try to find the locations of objects of a set of classes in the image. So we might try to mark all cars, all cats, all camels, and so on. As far as anyone knows, the right way to think about object detection is that we search a collection of windows in an image, apply an image classification method to each window, then resolve disputes between overlapping windows. How windows are to be chosen for this purpose is an active and quickly changing area of research. We will regard image classification as the key building block, and ignore the question of deciding which window to classify.

We have most of the pieces to build an image classifier. Architectural choices will make a difference to its performance. So will a series of tricks.

12.4.1 An Image Classification Architecture

We can now put together an image classifier. A convolutional layer receives image pixel values as input. The output is fed to a stack of convolutional layers, each feeding the next. The output of the final layer is fed to one or more fully connected layers, with one output per class. The whole is trained by batch gradient descent, or a variant, as above.

There are a number of architectural choices to make, which are typically made by experiment. The main ones are the choice of the number of convolutional layers; the choice of the number of slices in each layer; and the choice of stride for each convolutional layer. There are some constraints on the choice of stride. The first convolutional layer will tend to have stride 1, so that we see all the resolution of the image. But the outputs of that layer are likely somewhat correlated, because they depend on largely the same set of pixels. Later layers might have larger stride for this reason. In turn, this means the spatial dimensions of the representation will get smaller.

Notice that different image classification networks differ by relatively straightforward changes in architectural parameters. Mostly, the same thing will happen to these networks (variants of batch gradient descent on a variety of costs; dropout; evaluation). In turn, this means that we should use some form of specification language to put together a description of the architecture of interest. Ideally, in such an environment, we describe the network architecture, choose an optimization algorithm, and choose some parameters (dropout probability, etc.). Then the environment assembles the net, trains it (ideally, producing log files we can look at) and runs an evaluation. Several such environments exist.

12.4.2 Useful Tricks - 1: Preprocessing Data

It usually isn't possible to simply feed any image into the network. We want each image fed into the network to be the same size. We can achieve this either by resizing the image, or by cropping the image. Resizing might mean we stretch or squash some images, which likely isn't a great idea. Cropping means that we need to make a choice about where the crop box lies in the image. Practical systems quite often apply the same network to different croppings of the same image. For our purposes, we will assume that all the images we deal with have the same size.

It is usually wise to preprocess images before using them. This is because two images with quite similar content might have rather different pixel values. For example, compare image $\mathcal{I}$ and $1.5\mathcal{I}$. One will be brighter than the other, but nothing substantial about the image class will have changed. There is little point in forcing the network to learn something that we know already. There are a variety of preprocessing options, and different options have proven to be best for different problems. I will sketch some of the more useful ones.

You could **whiten pixel values**. You would do this for each pixel in the image grid independently. For each pixel, compute the mean value at that pixel across the training dataset. Subtract this, and divide the result by the standard deviation of the value at that pixel across the training dataset. Each pixel location in the resulting stack of images has mean zero and standard deviation one. Reserve the offset image (the mean at each pixel location) and the scale image (ditto, standard deviation) so that you can normalize test images.

You could **contrast normalize the image** by computing the mean and standard deviation of pixel values in each training (resp. test) image, then subtracting the mean from the image and dividing the result by the standard deviation.

You could **contrast normalize pixel values locally**. To do so, you compute a smoothed version of the image (convolve with a Gaussian, for insiders; everyone else should skip this paragraph, or perhaps search the internet). You can think of the result as a local estimate of the image mean. At each pixel, you subtract the smoothed value from the image value.

Useful Facts: 12.1 *Whitening a dataset*

For a dataset $\{\mathbf{x}\}$, compute:

- $\mathcal{U}$, the matrix of eigenvectors of $\mathsf{Covmat}(\{\mathbf{x}\})$;
- and $\mathsf{mean}(\{\mathbf{x}\})$.

Now compute $\{\mathbf{n}\}$ using the rule

$$\mathbf{n}_i = \mathcal{U}^T(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\})).$$

Then $\mathsf{mean}(\{\mathbf{n}\}) = \mathbf{0}$ and $\mathsf{Covmat}(\{\mathbf{n}\})$ is diagonal.
Now write Λ for the diagonal matrix of eigenvalues of $\mathsf{Covmat}(\{\mathbf{x}\})$ (so that $\mathsf{Covmat}(\{\mathbf{x}\})\mathcal{U} = \mathcal{U}\Lambda$). Assume that each of the diagonal entries of Λ is greater than zero (otherwise there is a redundant dimension in the data). Write λ_i for the i'th diagonal entry of Λ, and write $\Lambda^{-(1/2)}$ for the diagonal matrix whose i'th diagonal entry is $1/\sqrt{\lambda_i}$. Compute $\{\mathbf{z}\}$ using the rule

$$\mathbf{z}_i = \Lambda^{(-1/2)}\mathcal{U}(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\})).$$

We have that $\mathsf{mean}(\{\mathbf{z}\}) = 0$ and $\mathsf{Covmat}(\{\mathbf{z}\}) = \mathcal{I}$. The dataset $\{\mathbf{z}\}$ is often known as **whitened data**.

You could whiten the image as in section 12.1. It turns out this doesn't usually help all that much. Instead, you need to use **ZCA-whitening**. I will use the same notation as chapter 10.1, but I reproduce the useful facts box here as a reminder. Notice that, by using the rule

$$\mathbf{z}_i = \Lambda^{(-1/2)}\mathcal{U}(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\})),$$

we have rotated the data in the high dimensional space. In the case of images, this means that the image corresponding to $\mathbf{z}_i$ will likely not look like anything coherent. Furthermore, if there are very small eigenvalues, the scaling represented by $\Lambda^{-(1/2)}$ may present serious problems. But notice that the covariance matrix of a dataset is unaffected by rotation. We could choose a small non-negative constant ϵ, and use the rule

$$\mathbf{z}_i = \mathcal{U}^T(\Lambda + \epsilon\mathcal{I})^{(-1/2)}\mathcal{U}(\mathbf{x}_i - \mathsf{mean}(\{\mathbf{x}\}))$$

instead. The result looks significantly more like an image, and will have a covariance matrix that is the identity (or close, depending on the value of ϵ). This rule is ZCA whitening.

12.4.3 Useful Tricks - 2: Enhancing Training Data

Datasets of images are never big enough to show all effects accurately. This is because an image of a horse is still an image of a horse even if it has been through a small rotation, or has been resized to be a bit bigger or smaller, or has been cropped differently, and so on. There is no way to take account of these effects in the architecture of the network. Generally, a better approach is to expand the training dataset to include different rotations, scalings, and crops of images.

Doing so is relatively straightforward. You take each training image, and generate a collection of extra training images from it. You can obtain this collection by: resizing and then cropping the training image; using different crops of the same training image (assuming that training images are a little bigger than the size of image you will work with); rotating the training image by a small amount, resizing and cropping; and so on. You can't crop too much, because you need to ensure that the modified images are still of the relevant class, and an aggressive crop might cut out the horse, etc. When you rotate then crop, you need to be sure that no "unknown" pixels find their way into the final crop. All this means that only relatively small rescales, crops, rotations, etc. will work. Even so, this approach is an extremely effective way to enlarge the training set.

12.4.4 Useful Tricks - 3: Batch Normalization

There is good experimental evidence that large values of inputs to any layer within a neural network lead to problems. One source of the problem could be this. Imagine some input to some unit has a large absolute value. If the corresponding weight is relatively small, then one gradient step could cause the weight to change sign. In turn, the output of the unit will swing from one side of the RELU's non-linearity to the other. If this happens for too many units, there will be training problems because the gradient is then a poor prediction of what will actually happen to the output. So we should like to ensure that relatively few values at the input of any layer have large absolute values. We will build a new layer, sometimes called a **batch normalization layer**, which can be inserted between two existing layers.

Write $\mathbf{x}^b$ for the input of this layer, and $\mathbf{o}^b$ for its output. The output has the same dimension as the input, and I shall write this dimension d. The layer has two vectors of parameters, γ and β, each of dimension d. Write diag($\mathbf{v}$) for the matrix whose diagonal is $\mathbf{v}$, and with all other entries zero. Assume we know the mean ($\mathbf{m}$) and standard deviation ($\mathbf{s}$) of each component of $\mathbf{x}^b$, where the expectation is taken over all relevant data. The layer forms

$$\begin{aligned}\mathbf{x}^n &= [\mathrm{diag}(\mathbf{s}+\epsilon)]^{-1}\left(\mathbf{x}^b-\mathbf{m}\right)\\ \mathbf{o}^b &= [\mathrm{diag}(\gamma)]\,\mathbf{x}^n+\beta.\end{aligned}$$

Notice that the output of the layer is a differentiable function of γ and β. Notice also that this layer *could* implement the identity transform, if $\gamma = \mathrm{diag}(\mathbf{s}+\epsilon)$ and

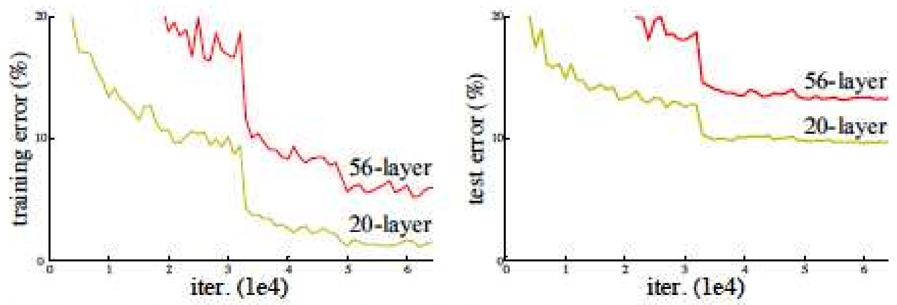


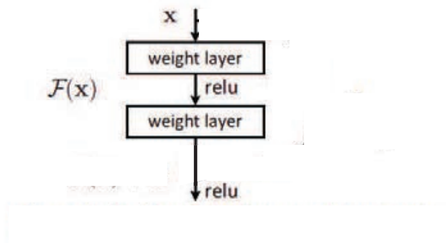
Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

FIGURE 12.13: *This figure from* Deep Residual Learning for Image Recognition *Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun, ICCV 2016, illustrates the difficulties presented by training a deep network.*

$\beta = \mathbf{m}$. We adjust the parameters in training to achieve the best performance. It can be helpful to think about this layer as follows. The layer rescales its input to have zero mean and unit standard deviation, then allows training to readjust the mean and standard deviation as required. In essence, we expect that large values encountered between layers are likely an accident of the difficulty training a network, rather than required for good performance.

The difficulty here is we don't know either $\mathbf{m}$ or $\mathbf{s}$, because we don't know the parameters used for previous layers. Current practice is as follows. First, start with $\mathbf{m} = \mathbf{0}$ and $\mathbf{s} = \mathbf{1}$ for each layer. Now choose a minibatch, and train the network using that minibatch. Once you have taken enough gradient steps and are ready to work on another minibatch, reestimate $\mathbf{m}$ as the mean of values of the inputs to the layer, and $\mathbf{s}$ as the corresponding standard deviations. Now obtain another minibatch, and proceed. Remember, γ and β are parameters that are trained, just like the others (using gradient descent, momentum, adagrad, or whatever). Once the network has been trained, one then takes the mean (resp. standard deviation) of the layer inputs over the training data for $\mathbf{m}$ (resp. $\mathbf{s}$). Most neural network implementation environments will do all the work for you. It is quite usual to place a batch normalization layer between each layer within the network.

There is a general agreement that batch normalization improves training, but some disagreement about the details. Experiments comparing two networks, one with batch normalization the other without, suggest that the same number of steps tends to produce a lower error rate when batch normalized. Some authors suggest that convergence is faster (which isn't quite the same thing). Others suggest that larger learning rates can be used.



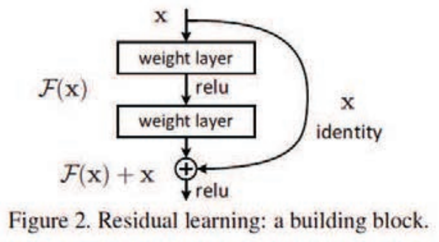

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu
Figure 2. Residual learning: a building block.

FIGURE 12.14: *This figure, which is revised from* Deep Residual Learning for Image Recognition *Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun, ICCV 2016, conveys the intention of a residual network.*

12.4.5 Useful Tricks - 4: Residual Networks

A randomly initialized deep network can so severely mangle its inputs that only a wholly impractical amount of training will cause the latest layers to do anything useful. As a result, there have been practical limits on the number of layers that can be stacked (Figure 12.13). One recent strategy for avoiding this difficulty is to build a **residual layer**. Figure 12.14 sketches the idea in the form currently best understood. Remember, $F(x)$ is a RELU. Our usual layers produce $\mathbf{x}^{l+1} = \mathcal{F}(\mathbf{x}^l; \theta) = F(\mathcal{W}\mathbf{x}^l + \mathbf{b})$ as its output. This layer could be anything, but is most likely a fully connected or a convolutional layers. Then we can replace this layer with one that produces

$$\begin{array}{rcl}\mathbf{x}^{l+1} & = & F(\mathbf{x}^l + \mathcal{W}_1\mathbf{q} + \mathbf{b}_1)\\ \mathbf{q} & = & F(\mathcal{W}_2\mathbf{x}^l + \mathbf{b}_2).\end{array}$$

It is usual, if imprecise, to think of this as producing an output that is $\mathbf{x} + \mathcal{F}(\mathbf{x}; \theta)$ — the layer passes on its input with a residual added to it. The point of all this is that, at least in principle, this residual layer can represent its output as a small offset on its input. If it is presented with large inputs, it can produce large outputs by passing on the input. Its output is also significantly less mangled by stacking layers, because its output is largely given by its input plus a non-linear function.

Very recently, an improvement on this strategy has surfaced, in *Identity Mappings in Deep Residual Networks* by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (which you can find on ArXiV using a search engine). Rather than use the expression above (corresponding to Figure 12.14), we use a layer that produces

$$\begin{array}{rcl}\mathbf{x}^{l+1} & = & \mathbf{x}^l + \mathcal{W}_1\mathbf{q} + \mathbf{b}_1\\ \mathbf{q} & = & F(\mathcal{W}_2F(\mathbf{x}^l) + \mathbf{b}_2).\end{array}$$

It is rather more informative to think of this as producing an output that is $\mathbf{x} + \mathcal{F}(\mathbf{x}; \theta)$ — the layer passes on its input with a residual added to it. There is good

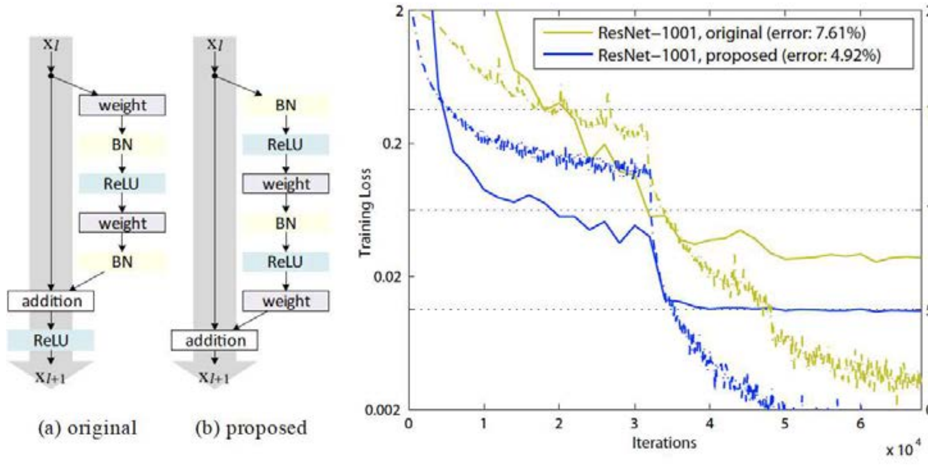


FIGURE 12.15: *This figure is revised from* Identity Mappings in Deep Residual Networks *by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (which you can find on ArXiV using a search engine). On the* **left***, (a) shows the original residual network architecture, also described in Figure 12.14; (b) shows the current best architecture. On the* **right***, train (dashed) and test (full) curves for the old and new architectures on CIFAR-10. Notice the significant improvement in performance.*

evidence that such layers can be stacked very deeply indeed (the paper I described uses 1001 layers to get under 5% error on CIFAR-10; beat that if you can!). One reason is that there are useful components to the gradient for each layer that do not get mangled by previous layers. You can see this by considering the Jacobian of such a layer with respect to its inputs. You will see that this Jacobian will have the form

$$\mathcal{J}_{x^{l+1};x^l} = (\mathcal{I} + \mathcal{M}_l)$$

where $\mathcal{I}$ is the identity matrix and $\mathcal{M}_l$ is a set of terms that depend on $\mathcal{W}$ and $\mathbf{b}$. Now remember that, when we construct the gradient at the k'th layer, we evaluate by multiplying a set of Jacobians corresponding to the layers above. This product in turn must look like

$$\mathcal{J}_{o;\theta^k} = \mathcal{J}_{o;\mathbf{x}^{k+1}} \mathcal{J}_{\mathbf{x}^{k+1};\theta^k} = (\mathcal{I} + \mathcal{M}_1 + \ldots)\mathcal{J}_{\mathbf{x}^{k+1};\theta^k}$$

which means that some components of the gradient at that layer do not get mangled by being passed through a sequence of poorly estimated Jacobians. One reason I am having trouble making a compelling argument explaining why this architecture is better is that the argument doesn't seem to be known in any tighter form (it certainly isn't to me). There is overwhelming evidence that the architecture is, in practice, better; it has produced networks that are (a) far deeper and (b) far more accurate than anything produced before. But why it works remains somewhat veiled.