Chapter 1

Background

This thesis combines work in several fields. We provide a background for the most important and relevant fields in this chapter. We first introduce the basics of deep learning, before defining the properties of Wavelet Transforms, and finally we introduce the Scattering Transform, the original inspiration for this thesis.

1.1 Supervised Machine Learning

Consider a sample space over inputs and labels $\mathcal{X} \times \mathcal{Y}$ and a data generating distribution $p_d ata$. Given a dataset of input-label pairs $\mathcal{D} = \{(x^{(n)}, y^{(n)})\}_{n=1}^N$ we would like to make predictions about $p_d ata(y|x)$ that generalize well to unseen data. A common way to do this is to build a parametric model to directly estimate this conditional probability. For example, regression asserts the data are distributed according to a function of the inputs plus a noise term ϵ:

$$y = f(x, \theta) + \epsilon \tag{1.1.1}$$

This noise is often modelled as a zero mean Gaussian random variable, $\epsilon \sim \mathcal{N}(0, \sigma^2)$, which means we can write:

$$p_{model}(y|x, \theta) = \mathcal{N}(y;\ f(x, \theta), \sigma^2) \tag{1.1.2}$$

with (θ, σ^2) are the parameters of the model.

We can find point estimates of the parameters by maximizing the likelihood of $p_{model}(y|x, \theta)$ (or equivalently, minimizing the KL-divergence between p_{model} and p_{data} $KL(p_{model}||p_{data})$).

As the data are all i.i.d., we can multiply individual likelihoods, and solve for θ:

$$\begin{aligned}\theta_{MLE} &= \arg\max_{\theta} p_{model}(y|x,\theta) && (1.1.3)\\ &= \arg\max_{\theta} \prod_{n=1}^{N} p_{model}(y^{(n)}|x^{(n)},\theta) && (1.1.4)\\ &= \arg\max_{\theta} \sum_{n=1}^{N} \log p_{model}(y^{(n)}|x^{(n)},\theta) && (1.1.5)\end{aligned}$$

Using the regression model from above, this becomes:

$$\begin{aligned}\theta_{MLE} &= \arg\max_{\theta} \sum_{n=1}^{N} \log p_{model}(y^{(n)}|x^{(n)},\theta) && (1.1.6)\\ &= \arg\max_{\theta} \left(-N\log\sigma - \frac{N}{2}\log(2\pi) - \sum_{n=1}^{N} \frac{\left(y^{(n)} - f(x^{(n)},\theta)\right)^2}{2\sigma^2} \right) && (1.1.7)\\ &= \arg\min_{\theta} \frac{1}{N}\sum_{n=1}^{N} \frac{\left(y^{(n)} - f(x^{(n)},\theta)\right)^2}{2} && (1.1.8)\end{aligned}$$

which gives us the well known result that we would like to find parameters that minimize the mean squared error (MSE) between observations y and predictions $\hat{y} = f(x,\theta)$.

For binary classification, ($y \in \{0,1\}$) instead of the model in (1.1.2) we have:

$$p_{model}(y|x,\theta) = \text{Ber}(y;\ \sigma(f(x,\theta))) \qquad (1.1.9)$$

where σ is the sigmoid function:

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad (1.1.10)$$

This expands naturally to multi-class classification ($y \in \{0,1\}^C$) by swapping the Bernoulli distribution for the categorical and the sigmoid for a softmax function, defined by:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{C} e^{z_k}} \qquad (1.1.11)$$

If we let $\hat{y}_i = \sigma_i(f(x,\theta))$, this makes (1.1.9):

$$\begin{aligned}p_{model}(y|x,\theta) &= \text{Cat}(y;\ \sigma(f(x,\theta))) && (1.1.12)\\ &= \prod_{c=1}^{C}\prod_{n=1}^{N} \left(\hat{y}_c^{(n)}\right)^{\mathbb{I}(y_c^{(n)}=1)} && (1.1.13)\end{aligned}$$

where $\mathbb{I}(x)$ is the indicator function. As $y_c^{(n)}$ is either 0 or 1, we remove the indicator function. Maximizing this likelihood to find the ML estimate for θ:

$$\begin{aligned}\theta_{MLE} &= \arg\min_{\theta} \prod_{c=1}^{C}\prod_{n=1}^{N}\left(\hat{y}_c^{(n)}\right)^{y_c^{(n)}} && (1.1.14)\\ &= \arg\min_{\theta} \frac{1}{N}\sum_{n=1}^{N}\sum_{c=1}^{C} y_c^{(n)}\log \hat{y}_c^{(n)} && (1.1.15)\end{aligned}$$

which we recognize as the cross-entropy between y and $\hat{y}$.

1.1.1 Priors on Parameters and Regularization

Maximum likelihood estimates for parameters, while straightforward, can often lead to overfitting. A common practice is to regularize learnt parameters θ by putting a prior over them. If we do not have any prior information about what we expect the parameters to be, it is still useful to put an uninformative prior on the weights. For example, if our weights are in the reals, a commonly used prior is a Gaussian.

Let us extend the regression example from above by saying we would like the prior on the weights θ to be a Gaussian, i.e. $p(\theta) = \mathcal{N}(0,\tau^2)$. The corresponding maximum a posteriori (MAP) estimate is then obtained by finding:

$$\theta_{MAP} = \arg\min_{\theta} \frac{1}{N}\sum_{n=1}^{N}\frac{\left(y^{(n)} - f(x^{(n)},\theta)\right)^2}{2} + \lambda||\theta||_2^2 \qquad (1.1.16)$$

where $\lambda = \sigma^2/\tau^2$, which is equivalent to minimizing the MSE with an ℓ_2 penalty on the parameters. λ is often called **weight decay** in the neural network literature, which we will also use in this thesis.

1.1.2 Loss Functions and Minimizing the Objective

It may be useful to rewrite (1.1.16) as an objective function on the parameters $J(\theta)$:

$$\begin{aligned}J(\theta) &= \frac{1}{N}\sum_{n=1}^{N}\frac{\left(y^{(n)} - f(x^{(n)},\theta)\right)^2}{2} + \lambda||\theta||_2^2 && (1.1.17)\\ &= L_{data}(y, f(x,\theta)) + L_{reg}(\theta) && (1.1.18)\end{aligned}$$

where L_{data} is the data loss defined, such as MSE or cross-entropy and L_{reg} is the regularization, such as ℓ_2 or ℓ_1 penalized loss.

Now $\theta_{MAP} = \arg\min J(\theta)$. Finding the minimum of the objective function is task-dependent and is often not straightforward. One commonly used technique is called *gradient descent* (GD). This is straightforward to do as it only involves calculating the gradient at

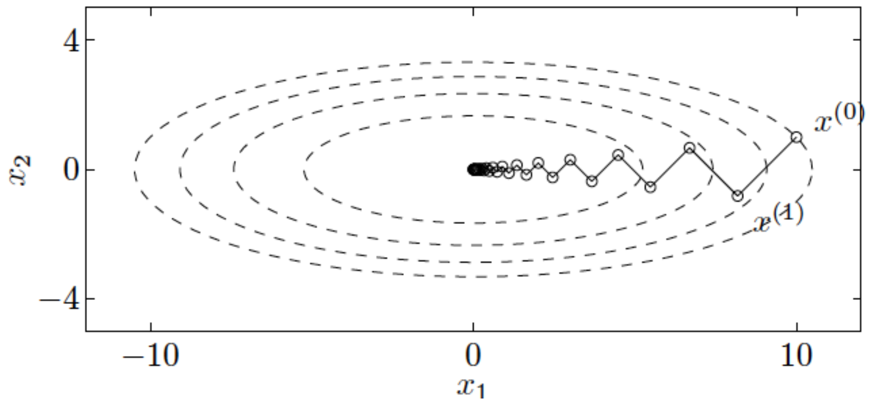

Figure 1.1: **Trajectory of gradient descent in an ellipsoidal parabola.** Some contour lines of the function $f(x) = 1/2\,(x_1^2 + 10x_2^2)$ and the trajectory of GD optimization using exact line search. This space has condition number 10, and shows the slow convergence of GD in spaces with largely different eigenvalues. Image taken from [1] Figure 9.2.

a given point and taking a small step in the direction of steepest descent. The difference equation defining this can be written as:

$$\theta_{t+1} = \theta_t - \eta \frac{\partial J}{\partial \theta} \tag{1.1.19}$$

Unsurprisingly, such a simple technique has limitations. In particular, it has a slow convergence rate when the condition number (ratio of largest to smallest eigenvalues) of the Hessian around the optimal point is large [1]. An example of this is shown in Figure 1.1. In this figure, the step size is chosen with exact line search, i.e.

$$\eta = \arg\min_s f(x + s\frac{\partial f}{\partial x}) \tag{1.1.20}$$

To truly overcome this problem, we must know the curvature of the objective function $\frac{\partial^2 J}{\partial \theta^2}$. An example optimization technique that uses the second order information is Newton's method. Such techniques sadly do not scale with size, as computing the Hessian is proportional to the number of parameters squared, and most neural networks have hundreds of thousands, if not millions of parameters. In this thesis, we only consider *first-order optimization* algorithms.

1.1.3 Stochastic Gradient Descent

Aside from the problems associated the curvature of the function $J(\theta)$, another common issue faced with the gradient descent of (1.1.19) is the cost of computing $\frac{\partial J}{\partial \theta}$. In particular, the first term:

$$
\begin{aligned}
L_{data}(y, f(x,\theta)) &= \mathbb{E}_{x,y\sim p_{data}}\left[L_{data}(y, f(x,\theta))\right] && (1.1.21)\\
&= \frac{1}{N}\sum_{n=1}^{N} L_{data}\left(y^{(n)}, f(x^{(n)},\theta)\right) && (1.1.22)
\end{aligned}
$$

involves evaluating the entire dataset at the current values of θ. As the training set size grows into the thousands or millions of examples, this approach becomes prohibitively slow.

(1.1.21) writes the data loss as an expectation, hinting at the fact that we can remedy this problem by using fewer samples $N_b < N$ to evaluate L_{data}. This variation is called Stochastic Gradient Descent (SGD).

Choosing the batch size is a hyperparameter choice that we must think carefully about. Setting the value very low, e.g. $N_b = 1$ can be advantageous as the noisy estimates for the gradient have a regularizing effect on the network [2]. Increasing the batch size to larger values allows you to easily parallelize computation as well as increasing your accuracy for the gradient, allowing you to take larger step sizes [3]. A good initial starting point is to set the batch size to about 100 samples and increase/decrease from there [4].

1.1.4 Gradient Descent and Learning Rate

The step size parameter, η in (1.1.19) is commonly referred to as the learning rate. Choosing the right value for the learning rate is key. Unfortunately, the line search algorithm in (1.1.20) would be too expensive to compute for neural networks (as would involve evaluating the function several times at different values), each of which takes about as long as calculating the gradients themselves. Additionally, as the gradients are typically estimated over a mini-batch and are hence noisy there may be little added benefit in optimizing the step sizes in the estimated direction.

Figure 1.2 illustrates the effect the learning rate can have over a contrived convex example. Optimizing over more complex loss surfaces only exacerbates the problem. Sadly, choosing the initial learning rate is 'more of an art than a science' [4], but [5], [6] have some tips on what to set this at. We have found in our work that searching for a large learning rate that causes the network to diverge and reducing it hence can be a good search strategy. This agrees with Section 1.5 of [7] which states that for regions of the loss space which are roughly quadratic, $\eta_{max} = 2\eta_{opt}$ and any learning rate above $2\eta_{opt}$ causes divergence.

On top of the initial learning rate, the convergence of SGD methods require:

$$
\begin{aligned}
\sum_{t=1}^{\infty}\eta_t &\to \infty && (1.1.23)\\
\sum_{t=1}^{\infty}\eta_t^2 &= M && (1.1.24)
\end{aligned}
$$

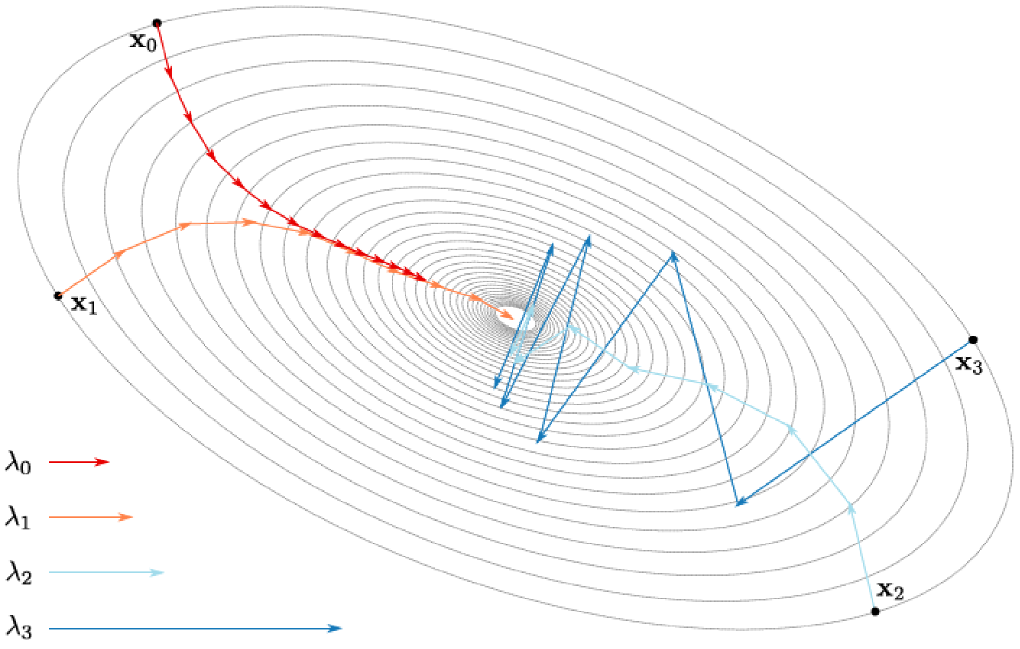

Figure 1.2: **Trajectories of SGD with different initial learning rates.** This figure illustrates the effect the step size has over the optimization process by showing the trajectory for $\eta = \lambda_i$ from equivalent starting points on a symmetric loss surface. Increasing the step size beyond λ_3 can cause the optimization procedure to diverge. Image taken from [8] Figure 2.7.

where M is finite. Choosing how to do this also contains a good amount of artistry, and there is no one scheme that works best. A commonly used greedy method is to keep the learning rate constant until the training loss stabilizes and then to enter the next phase of training by setting $\eta_{k+1} = \gamma\eta_k$ where γ is a decay factor. Choosing γ and the thresholds for triggering a step however must be chosen by monitoring the training loss curve and trial and error.

1.1.5 Momentum and Adam

One simple and very popular modification to SGD is to add *momentum*. Momentum accumulates past gradients with an exponentially moving average and continues to move in their direction. The name comes from the analogy of finding a minimum of a function to rolling a ball over a loss surface – any new force (newly computed gradients) must overcome the past motion of the ball. To do this, we create a *velocity* variable v_t and modify (1.1.19) to be:

$$\begin{aligned} v_{t+1} &= \alpha v_t - \eta_k \frac{\partial J}{\partial \theta} && (1.1.25) \\ \theta_{t+1} &= \theta_t + v_{t+1} && (1.1.26) \\ & && (1.1.27) \end{aligned}$$

where $0 \leq \alpha < 1$ is the momentum term indicating how quickly to 'forget' past gradients.

Another popular modification to SGD is the adaptive learning rate technique Adam [9]. There are several other adaptive schemes such as AdaGrad [10] and AdaDelta [11], but they are all quite similar, and Adam is often considered the most robust of the three [4]. The goal of all of these adaptive schemes is to take larger update steps in directions of low variance, helping to minimize the effect of large condition numbers we saw in **??**. Adam does this by keeping track of the first m_t and second v_t moments of the gradients:

$$\begin{aligned} g_{t+1} &= \frac{\partial J}{\partial \theta} && (1.1.28) \\ m_{t+1} &= \beta_1 m_t + (1-\beta_1) g_{t+1} && (1.1.29) \\ v_{t+1} &= \beta_2 m_t + (1-\beta_2) g_{t+1} && (1.1.30) \end{aligned}$$

where $0 \leq \beta_1, \beta_2 < 1$. Note the similarity between updating the mean estimate in (1.1.29) and the velocity term in (1.1.25)[1]. The parameters are then updated with:

$$\theta_{t+1} = \theta_t - \eta \frac{m_{t+1}}{\sqrt{v_{t+1}} + \epsilon} \quad (1.1.31)$$

where ϵ is a small value to avoid dividing by zero.

1.2 Neural Networks

1.2.1 The Neuron and Single Layer Neural Networks

The neuron, shown in Figure 1.3 is the core building block of Neural Networks. It takes the dot product between an input vector $\mathbf{x} \in \mathbb{R}^D$ and a weight vector $\mathbf{w}$, before applying a chosen nonlinearity, g. I.e.

$$y = g(\langle \mathbf{x}, \mathbf{w} \rangle) = g\left(\sum_{i=0}^{D} x_i w_i \right) \quad (1.2.1)$$

[1]The m_{t+1} and v_{t+1} terms are then bias-corrected as they are biased towards zero at the beginning of training. We do not include this for conciseness.

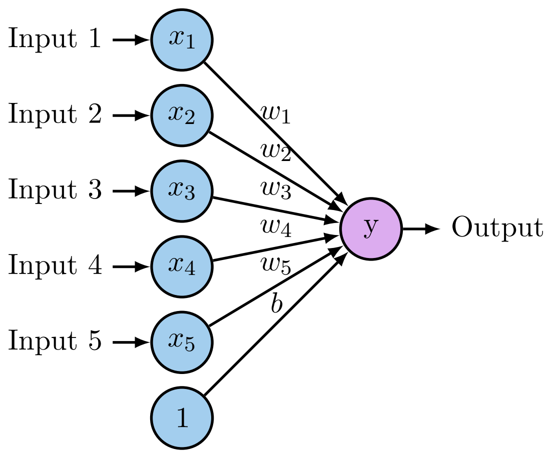

Figure 1.3: **A single neuron.** The neuron is composed of inputs x_i, weights w_i (and a bias term), as well as an activation function. Typical activation functions include the sigmoid function, tanh function and the ReLU

where we have used the shorthand $b = w_0$ and $x_0 = 1$. Also note that we will use the neural network common practice of calling the *weights* w, compared to the parameters θ we have been discussing thus far.

Typical nonlinear functions g are the sigmoid function (already presented in (1.1.10)), but also common are the tanh and ReLU functions:

$$\begin{aligned} \tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} && (1.2.2) \\ \text{ReLU}(x) &= \max(x, 0) && (1.2.3) \end{aligned}$$

See Figure 1.4 for plots of these. The original Rosenblatt perceptron [12] used the Heaviside function $\text{H}(x) = \mathbb{I}(x > 0)$.

Note that if $\langle \mathbf{w}, \mathbf{w} \rangle = 1$ then $\langle \mathbf{x}, \mathbf{w} \rangle$ is the distance from the point $\mathbf{x}$ to the hyperplane with normal $\mathbf{w}$. With general $\mathbf{w}$ this can be thought of as a scaled distance. Thus, the weight vector $\mathbf{w}$ defines a hyperplane in $\mathbb{R}^D$ which splits the space into two. The choice of nonlinearity then affects how points on each side of the plane are treated. For a sigmoid, points far below the plane get mapped to 0 and points far above the plane get mapped to 1 (with points near the plane having a value of 0.5). For tanh nonlinearities, these points get mapped to -1 and 1. For ReLU nonlinearities, every point below the plane ($\langle \mathbf{x}, \mathbf{w} \rangle < 0$) gets mapped to zero and every point above the plane keeps its inner product value.

Nearly all modern neural networks use the ReLU nonlinearity and it has been credited with being a key reason for the recent surge in deep learning success [13], [14]. In particular:

1. It is less sensitive to initial conditions as the gradients that backpropagate through it will be large even if x is large. A common observation of sigmoid and tanh non-linearities was that their learning would be slow for quite some time until the neurons came out

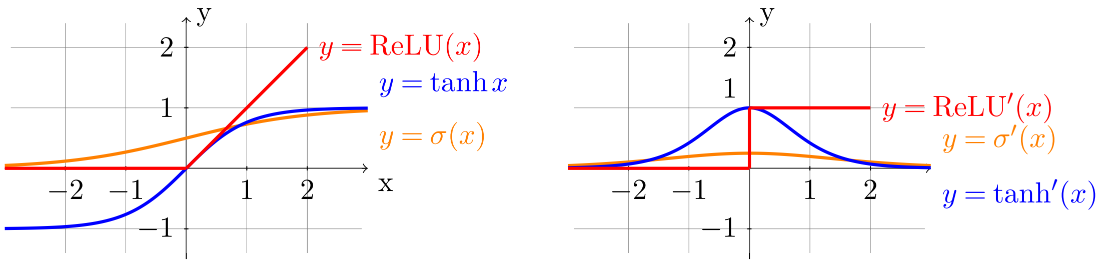

Figure 1.4: **Common Neural Network nonlinearities and their gradients.** The sigmoid, tanh and ReLU nonlinearities are commonly used activation functions for neurons. Note the different properties. In particular, the tanh and sigmoid have the nice property of being smooth but can have saturation when the input is either largely positive or largely negative, causing little gradient to flow back through it. The ReLU does not suffer from this problem, and has the additional nice property of setting values to exactly 0, making a sparser output activation.

of saturation, and then their accuracy would increase rapidly before levelling out again at a minimum [15]. The ReLU, on the other hand, has constant gradient.

2. It promotes sparsity in outputs, by setting them to a hard 0. Studies on brain energy expenditure suggest that neurons encode information in a sparse manner. [16] estimates the percentage of neurons active at the same time to be between 1 and 4%. Sigmoid and tanh functions will typically have *all* neurons firing, while the ReLU can allow neurons to fully turn off.

1.2.2 Multilayer Perceptrons

As mentioned in the previous section, a single neuron can be thought of as a separating hyperplane with an activation that maps the two halves of the space to different values. Such a linear separator is limited, and famously cannot solve the XOR problem [17]. Fortunately, adding a single hidden layer like the one shown in Figure 1.5 can change this, and it is provable that with an infinitely wide hidden layer, a neural network can approximate any function [18], [19].

The forward pass of such a network with one hidden layer of H units is:

$$h_i = g\left(\sum_{j=0}^{D} x_j w_{ij}^{(1)}\right) \tag{1.2.4}$$

$$y = \sum_{k=0}^{H} h_k w_k^{(2)} \tag{1.2.5}$$

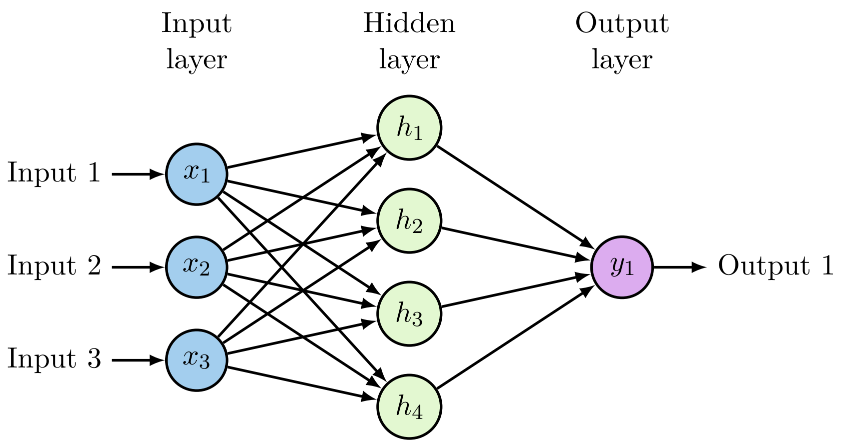

Figure 1.5: **Multi-layer perceptron.** Expanding the single neuron from Figure 1.3 to a network of neurons. The internal representation units are often referred to as the *hidden layer* as they are an intermediary between the input and output.

where $w^{(l)}$ denotes the weights for the l-th layer, of which Figure 1.5 has 2. Note that these individual layers are often called *fully connected* as each node in the previous layer affects every node in the next.

If we were to expand this network to have L such fully connected layers, we could rewrite the action of each layer in a recursive fashion:

$$Y^{(l+1)} = W^{(l+1)} X^{(l)} \tag{1.2.6}$$

$$X^{(l+1)} = g\left(Y^{(l+1)}\right) \tag{1.2.7}$$

where W is now a weight matrix, acting on the vector of previous layer's outputs $X^{(l)}$. As we are now considering every layer an input to the next stage, we have removed the h notation, and added the superscript (l) to define the depth. $X^{(0)}$ is the network input and $Y^{(L)}$ is the network output. Let us say that the output has C nodes, and a hidden layer $X^{(l)}$ has C_l nodes.

1.2.3 Backpropagation

It is important to truly understand backpropagation when designing neural networks, so we describe the core concepts now for a neural network with L layers.

The delta rule, initially designed for networks with no hidden layers [20], was expanded to what we now consider *backpropagation* in [21]. While backpropagation is conceptually just the application of the chain rule, Rumelhart, Hinton, and Williams successfully updated the delta rule to networks with hidden layers, laying a key foundational step for deeper networks.

With a deep network, calculating $\frac{\partial J}{\partial w}$ may not seem particularly obvious if w is a weight in one of the earlier layers. We need to define a rule for updating the weights in all L layers of the network, $W^{(1)}, W^{(2)}, \ldots W^{(L)}$ however, only the final set $W^{(L)}$ are connected to the objective function J.

1.2.3.1 Regression Loss

Let us start with writing down the derivative of J with respect to the network output $Y^{(L)}$ using the regression objective function (1.1.8). As we now have two superscripts, one for the sample number and one for the layer number, we combine them into a tuple of superscripts.

$$\begin{aligned}
\frac{\partial J}{\partial Y^{(L)}} &= \frac{\partial}{\partial Y^{(L)}} \left(\frac{1}{N} \sum_{n=1}^{N_b} \frac{1}{2} \left(y^{(n)} - Y^{(L,n)} \right)^2 \right) && (1.2.8)\\
&= \frac{1}{N} \sum_{n=1}^{N_b} \left(Y^{(L,n)} - y^{(n)} \right) && (1.2.9)\\
&= e \in \mathbb{R} && (1.2.10)
\end{aligned}$$

where we have used the fact that for the regression case, $y^{(n)}, Y^{(L,n)} \in \mathbb{R}$.

1.2.3.2 Classification Loss

For the classification case (1.1.15), let us keep the output of the network $Y^{(L,n)} \in \mathbb{R}^C$ and define an intermediate value $\hat{y}$ the softmax applied to this vector $\hat{y}_c^{(n)} = \sigma_c \left(Y^{(L,n)} \right)$. Note that the softmax is a vector valued function going from $\mathbb{R}^C \to \mathbb{R}^C$ so has a jacobian matrix $S_{ij} = \frac{\partial \hat{y}_i}{\partial Y_j^{(L)}}$ with values:

$$S_{ij} = \begin{cases} \sigma_i (1 - \sigma_j) & \text{if } i = j \\ -\sigma_i \sigma_j & \text{if } i \neq j \end{cases} \tag{1.2.11}$$

Now, let us return to (1.1.15) and find the derivative of the objective function to this intermediate value $\hat{y}$:

$$\begin{aligned}
\frac{\partial J}{\partial \hat{y}} &= \frac{\partial}{\partial \hat{y}} \left(\frac{1}{N} \sum_{n=1}^{N_b} \sum_{c=1}^{C} y_c^{(n)} \log \hat{y}_c^{(n)} \right) && (1.2.12)\\
&= \frac{1}{N} \sum_{n=1}^{N_b} \sum_{c=1}^{C} \frac{y_c^{(n)}}{\hat{y}_c^{(n)}} && (1.2.13)\\
&= d \in \mathbb{R}^C && (1.2.14)
\end{aligned}$$

Note that unlike (1.2.10), this derivative is vector valued. To find $\frac{\partial J}{\partial Y^{(L)}}$ we use the chain rule. It is easier to find the partial derivative with respect to one node in the output first, and then expand from here. I.e.:

$$\begin{aligned}\frac{\partial J}{\partial Y_j^{(L)}} &= \sum_{i=1}^{C} \frac{\partial J}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial Y_j^{(L)}} && (1.2.15)\\ &= S_j^T d && (1.2.16)\end{aligned}$$

where S_j is the jth column of the jacobian matrix S. It becomes clear now that to get the entire vector derivative for all nodes in $Y^{(L)}$, we must multiply the transpose of the jacobian matrix with the error term from (1.2.14):

$$\frac{\partial J}{\partial Y^{(L)}} = S^T d \qquad (1.2.17)$$

1.2.3.3 Final Layer Weight Gradient

Let us continue by assuming $\frac{\partial J}{\partial Y^{(L)}}$ is vector valued as was the case with classification. For regression, it is easy to set $C = 1$ in the following to get the necessary results. Additionally for clarity, we will drop the layer superscript in the intermediate calculations.

We call the gradient for the final layer weights the *update* gradient. It can be computed by the chain rule again:

$$\begin{aligned}\frac{\partial J}{\partial W_{ij}} &= \frac{\partial J}{\partial Y_i} \frac{\partial Y_i}{\partial W_{ij}} + 2\lambda W_{ij} && (1.2.18)\\ &= \frac{\partial J}{\partial Y_i} X_j + 2\lambda W_{ij} && (1.2.19)\end{aligned}$$

where the second term in the above two equations comes from the regularization loss that is added to the objective. The gradient of the entire weight matrix is then:

$$\begin{aligned}\frac{\partial J}{\partial W^{(L)}} &= \frac{\partial J}{\partial \hat{y}} X^T + 2\lambda W && (1.2.20)\\ &= S^T d \left(X^{(L-1)}\right)^T + 2\lambda W^{(L)} \in \mathbb{R}^{C \times C_{L-1}} && (1.2.21)\end{aligned}$$

1.2.3.4 Final Layer Passthrough Gradient

Additionally, we want to find the *passthrough* gradients of the final layer $\frac{\partial J}{\partial X^{(L-1)}}$. In a similar fashion, we first find the gradient with respect to individual elements in $X^{(L-1)}$ before generalizing to the entire vector:

$$\begin{aligned}\frac{\partial J}{\partial X_i} &= \sum_{j=1}^{C} \frac{\partial J}{\partial Y_j}\frac{\partial Y_j}{\partial X_i} && (1.2.22)\\ &= \sum_{j=1}^{C} \frac{\partial J}{\partial Y_j} W_{j,i} && (1.2.23)\\ &= W_i^T \frac{\partial J}{\partial Y} && (1.2.24)\\ & && (1.2.25)\end{aligned}$$

where W_i is the ith column of W. Thus

$$\begin{aligned}\frac{\partial J}{\partial X^{(L-1)}} &= \left(W^{(L)}\right)^T \frac{\partial J}{\partial Y^{(L)}} && (1.2.26)\\ &= \left(W^{(L)}\right)^T S^T d && (1.2.27)\end{aligned}$$

This passthrough gradient then can be used to update the next layer's weights by repeating subsubsection 1.2.3.3 and subsubsection 1.2.3.4.

1.2.3.5 General Layer Update

The easiest way to handle this flow of gradients, and the basis for most automatic differentiation packages, is the block definition shown in Figure 1.6. For all neural network components (even if they do not have weights), the operationg must not only be able to calculate the forward pass $y = f(x, w)$ given weights w and inputs x, but also calculate the *update* and *passthrough* gradients $\frac{\partial J}{\partial w}, \frac{\partial J}{\partial x}$ given an input gradient $\frac{\partial J}{\partial y}$. The input gradient will have the same shape as y as will the update and passthrough gradients match the shape of w and x. This way, gradients for the entire network cam be computed in an iterative fashion starting at the loss function and moving backwards.

1.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a special type of of Neural Network where the weights of the fully connected layer are shared across the layer. In this way, a neuron at a given layer is only affected by nodes from the previous layer in a given neighbourhood rather than every node.

First popularized in 1998 by LeCun et. al in [22], the convolutional layer was introduced to build invariance with respect to translations, as well as reduce the parameter size of early neural networks for pattern recognition. The idea of having a locally receptive field had already been shown to be a naturally occurring phenomen by Hubel and Wiesel [23]. They did not become popular immediately, and another spatially based keypoint extractor,

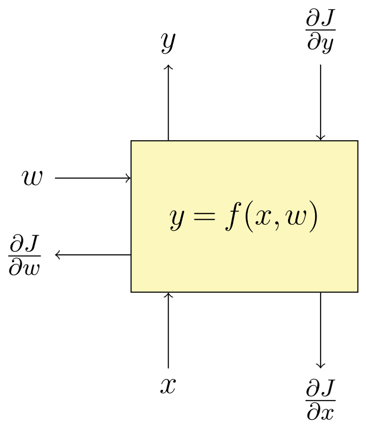
y
$\frac{\partial J}{\partial y}$
w
$y = f(x,w)$
$\frac{\partial J}{\partial w}$
x
$\frac{\partial J}{\partial x}$

Figure 1.6: **General block form for autograd.** All neural network functions need to be able to calculate the forward pass $y = f(x,w)$ as well as the update and passthrough gradients $\frac{\partial J}{\partial w}, \frac{\partial J}{\partial x}$. Backpropagation is then easily done by allowing data to flow backwards through these blocks from the loss.

SIFT [24], was the mainstay of detection systems until the AlexNet CNN [25] won the 2012 ImageNet challenge [26] by a large margin over the next competitors who used SIFT and Support Vector Machines [27]. This CNN had 5 convolutional layers followed by 3 fully connected layers.

We will briefly describe the convolutional layer, as well as many other layers that have become popular in the past few years.

1.3.1 Convolutional Layers

In the presentation of neural networks so far, we have considered column vectors $X^{(l)}, Y^{(l)} \in \mathbb{R}^{C_l}$. Convolutional layers for image analysis have a different format. In particular, the spatial component of the input is preserved.

Let us first consider the definition of 2-D convolution for single channel images:

$$\begin{aligned} y[\mathbf{n}] = (x * h)[\mathbf{n}] &= \sum_{\mathbf{k}} x[\mathbf{k}]h[\mathbf{n}-\mathbf{k}] && (1.3.1) \\ &= \sum_{k_1,k_2} x[k_1,k_2]h[n_1-k_1, n_2-k_2] && (1.3.2) \end{aligned}$$

where the sum is done over the support of h. For an input $x \in \mathbb{R}^{H\times W}$ and filter $h \in \mathbb{R}^{K_1\times K_2}$ the output has spatial support $y \in \mathbb{R}^{H+K_1-1\times W+K_2-1}$.

In the context of convolutional layers, this filter h is a *matched filter* that gives its largest output when the input contains h. If the input has shapes similar to h in many locations, each of these locations in y will also have large outputs.

It is not enough to only have a single matched filter, often we would like to have a bank of them, each sensitive to different shapes. For example, if h_1 was sensitive to horizontal edges, we may also want to detect vertical and diagonal edges. Without specifying what each of the filters do, we can however specify that we would like to detect C different shapes over the spatial extent of an input.

This then means that we have C output channels:

$$\begin{aligned} y_1[\mathbf{n}] &= (x * h_1)[\mathbf{n}] \\ y_2[\mathbf{n}] &= (x * h_1)[\mathbf{n}] \\ &\vdots \\ y_C[\mathbf{n}] &= (x * h_1)[\mathbf{n}] \end{aligned}$$

If we stack red, green and blue input channels on top of each other, we have a 3-dimensional array $x \in \mathbb{R}^{C\times H\times W}$ with $C = 3$.[2] In a CNN layer, each filter h is 3 dimensional with spatial extent exactly equal to C. The *convolution* is done over the remaining two dimensions and the C outputs are summed at each pixel location. This makes (1.3.1):

$$y[\mathbf{n}] = \sum_{c=1}^{C}\sum_{\mathbf{k}} x[\mathbf{k}]h[c, \mathbf{n}-\mathbf{k}] \tag{1.3.3}$$

Again, we would like to have many matched filters to find different shapes in the previous layer, so we repeat Equation 1.3.3 F times and stack the output to give $y \in \mathbb{R}^{F\times H\times W}$:

$$y[f, \mathbf{n}] = \sum_{c=1}^{C}\sum_{\mathbf{k}} x[c, \mathbf{k}]h_f[c, \mathbf{n}-\mathbf{k}] \tag{1.3.4}$$

After a convolutional layer, we can then apply a pointwise nonlinearity g to each output location in y. Revisiting (1.2.6) and (1.2.7), we can rewrite this for a convolutional layer at depth l with C_l input and C_{l+1} output channels:

$$Y^{(l+1)}[f, \mathbf{n}] = \sum_{c=1}^{C_l} X^{(l)}[c, \mathbf{n}] * h_f^{(l)}[c, \mathbf{n}] \qquad \text{for } 1 \le f \le C_{l+1} \tag{1.3.5}$$

$$X^{(l+1)}[f, \mathbf{n}] = g\left(Y^{(l)}[f, \mathbf{n}]\right) \tag{1.3.6}$$

This is shown in Figure 1.7.

[2] In deep learning literature, there is not consensus about whether to stack the outputs with the channel first ($\mathbb{R}^{C\times H\times W}$) or last ($\mathbb{R}^{H\times W\times C}$). The latter is more common in Image Processing for colour and spectral images, however the former is the standard for the deep learning framework we use – PyTorch [28], so we use this in this thesis.

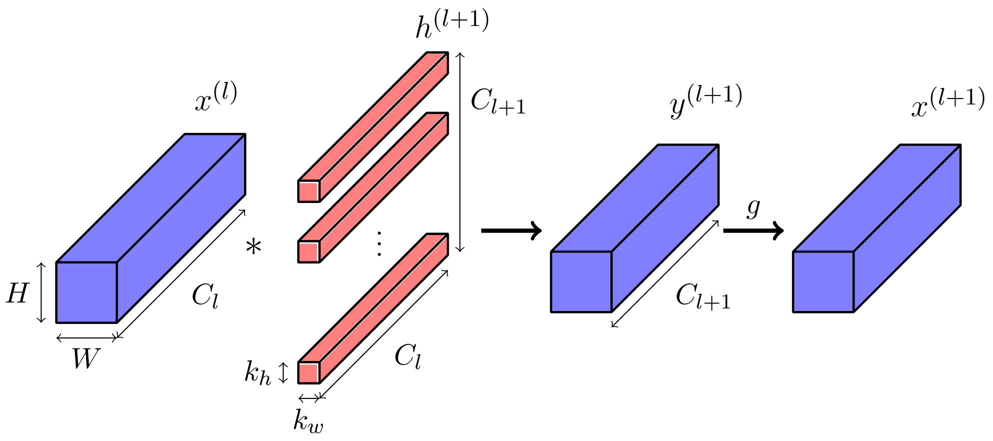

Figure 1.7: **A convolutional layer.** A convolutional layer followed by a nonlinearity g. The previous layer's activations are convolved with a bank of C_{l+1} filters, each of which has spatial size $k_h \times k_w$ and depth C_l. Note that there is no convolution across the channel dimension. Each filter produces one output channel in $y^{(l+1)}$.

1.3.1.1 Padding and Stride

Regular 2-D convolution expands the input from size $H \times W$ to $(H + K_H - 1) \times (W + K_W - 1)$. In convolutional layers in neural networks, it is often desirable and common to have the same output size as input size. This is achieved by taking the central $H \times W$ outputs. We call this *same-size convolution*. Another option commonly used is to only evaluate the kernels where they fully overlap the input signal, causing a reduction in the output size to $(H - K_H + 1) \times (W - K_W + 1)$. This is called *valid convolution* and was used in the original LeNet-5 [22].

Signal extension is by default *zero padding*, and most deep learning frameworks have no ability to choose other padding schemes as part of their convolution functions. Other padding such as *symmetric padding* can be achieved by expanding the input signal before doing a valid convolution.

Stride is a commonly used term in deep learning literature. A stride of 2 means that we evaluate the filter kernel at every other sample point. In signal processing, this is simply called decimation.

1.3.1.2 Gradients

To get the update and passthrough gradients for the convolutional layer we will need to expand (1.3.5). Again we will drop the layer superscripts for clarity:

$$Y[f, n_1, n_2] = \sum_{c=1}^{C} \sum_{k_1} \sum_{k_2} x[c, k_1, k_2] h_f[c, n_1 - k_1, n_2 - k_2] \tag{1.3.7}$$

It is clear from this that a single activation $X[c,n_1,n_2]$ affects many output values. In particular, the derivative for an activation in Y w.r.t. an activation in X is the sum of all the chain rule applied to all these positions:

$$\frac{\partial Y_{f,n_1,n_2}}{\partial X_{c,k_1,k_2}} = h_f[c, n_1-k_1, n_2-k_2] \tag{1.3.8}$$

and the derivative from the loss is then:

$$\begin{aligned}\frac{\partial J}{\partial X_{c,k_1,k_2}} &= \sum_f \sum_{n_1} \sum_{n_2} \frac{\partial J}{\partial Y_{f,n_1,n_2}} \frac{\partial Y_{f,n_1,n_2}}{\partial X_{c,k_1,k_2}} && (1.3.9)\\ &= \sum_f \sum_{n_1} \sum_{n_2} \frac{\partial J}{\partial Y_{f,n_1,n_2}} h_f[c,n_1-k_1,n_2-k_2] && (1.3.10)\end{aligned}$$

Now we let $\Delta Y[f,n_1,n_2] = \frac{\partial J}{\partial Y_{f,n_1,n_2}}$ be the passthrough gradient signal from the next layer, and $\tilde{h}_\alpha[\beta,\gamma,\delta] = h_\beta[\alpha,-\gamma,-\delta]$ be a set of filters that have been mirror imaged in the spatial domain and had their channel and filter number ordering swapped. Combining these two and subbing into (1.3.10) we get the *passthrough gradient* for the convolutional layer:

$$\begin{aligned}\frac{\partial J}{\partial X_{c,k_1,k_2}} &= \sum_f \sum_{n_1} \sum_{n_2} \Delta Y[f,n_1,n_2]\tilde{h}_c[f,k_1-n_1,k_2-n_2] && (1.3.11)\\ &= \sum_f \Delta Y[f,\mathbf{n}] * \tilde{h}_c[f,\mathbf{n}] && (1.3.12)\end{aligned}$$

which is the same as (1.3.5). I.e. we can backpropagate the gradients through a convolutional block by mirror imaging the filters, transposing them in the channel and filter dimensions, and doing a forward convolutional layer with $\tilde{h}$ applied to ΔY. Similarly, we find the *update gradients* to be:

$$\begin{aligned}\frac{\partial J}{\partial h_{f,c,k_1,k_2}} &= \sum_{n_1} \sum_{n_2} \frac{\partial J}{\partial Y_{f,n_1,n_2}} \frac{\partial Y_{f,n_1,n_2}}{\partial h_{f,c,k_1,k_2}} && (1.3.13)\\ &= \sum_{n_1} \sum_{n_2} \Delta Y[f,n_1,n_2] X[x,n_1-k_1,n_2-k_2] && (1.3.14)\\ &= (\Delta Y[f,\mathbf{n}] \star X[c,\mathbf{n}])\,[k_1,k_2] && (1.3.15)\end{aligned}$$

where $\star$ is the cross-correlation operation.

1.3.2 Pooling

Pooling layers are common in CNNs where we want to reduce the spatial size. As we go deeper into a CNN, it is common for the spatial size of the activation to decrease, and the channel dimension to increase. The C_l values at a given spatial location can then be thought of as a feature vector describing the presence of shapes in a given area in the input image.

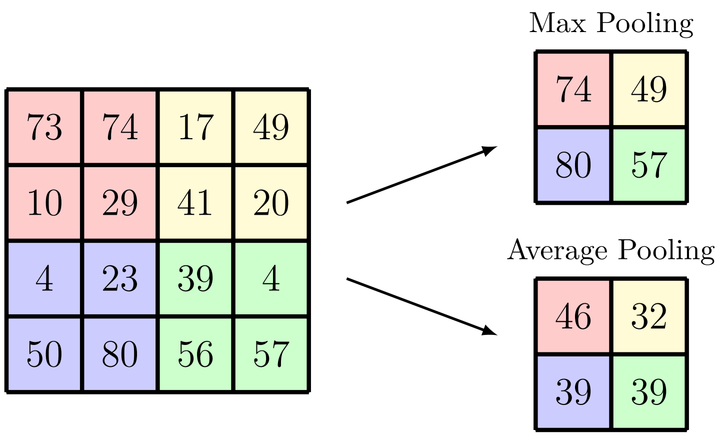

Figure 1.8: **Max vs Average 2×2 pooling.**

Pooling is useful to add some invariance to smaller shifts when downsampling. It is often done over small spatial sizes, such as 2×2 or 3×3. Invariance to larger shifts can be built up with multiple pooling (and convolutional) layers.

Two of the most common pooling techniques are *max pooling* and *average pooling*. Max pooling takes the largest value in its spatial area, whereas average pooling takes the mean. A visual explanation is shown in Figure 1.8. Note that pooling is typically a spatial operation, and only in rare cases is done over the channel dimension.

A review of pooling methods in [29] found them both to perform equally well. While max pooling was the most popular in earlier state of the art networks [25], [30], there has been a recent trend towards using average pooling [31] or even to do away with pooling altogether in favour of strided convolutions (this idea was originally proposed in [32] and used notably in [33]–[35]).

1.3.3 Dropout

Dropout is a particularly harsh regularization scheme that randomly turns off or zeros out neurons in a neural network [36], [37]. Each neuron has probability p of having its value set to 0 during training time, forcing the network to be more general and preventing 'co-adaption' of neurons. The main explanation given in [37] is that dropout averages over several 'thinner' models.

During test time, dropout is typically turned off, but can still be used to get an estimate on the uncertainty of the network by averaging over several runs [38].

1.3.4 Batch Normalization

Batch normalization proposed in [39] is a conceptually simple technique. Despite this, it has become very popular and has been found to be very useful to train *deeper* CNNs.

Batch Normalization rescales CNN activations by channel. Define the mean and standard deviations for a channel across the entire dataset as:

$$\mu_c = \frac{1}{N}\sum_{\mathbf{n}} X^{(n)}[c,\mathbf{n}] \tag{1.3.16}$$

$$\sigma_c^2 = \frac{1}{N}\sum_{\mathbf{n}} \left(X^{(n)}[c,\mathbf{n}]\right)^2 - \mu_c^2 \tag{1.3.17}$$

where $\mu,\sigma \in \mathbb{R}^C$. Batch norm removes the natural mean and variance of the data, scales the data by a learnable gain γ, and shifts it to a learnable position β, with $\gamma,\beta \in \mathbb{R}^C$:

$$Y[c,\mathbf{n}] = \frac{X[c,\mathbf{n}] - \mu_c}{\sigma_c + \epsilon}\gamma_c + \beta_c \tag{1.3.18}$$

where ϵ is a small value to avoid dividing by 0.

Of course, during training, we do not have access to the dataset μ,σ and these values must be estimated from the batch statistics. A typical practice is to keep an exponential moving average estimate of these values to give us $\tilde{\mu},\tilde{\sigma}$.

The passthrough and update gradients are:

$$\frac{\partial J}{\partial X_{c,n_1,n_2}} = \frac{\partial J}{\partial Y_{c,n_1,n_2}} \frac{\gamma}{\sigma+\epsilon} \tag{1.3.19}$$

$$\frac{\partial J}{\partial \beta_c} = \sum_{\mathbf{n}} \frac{\partial J}{\partial Y_{c,\mathbf{n}}} \tag{1.3.20}$$

$$\frac{\partial J}{\partial \gamma_c} = \sum_{\mathbf{n}} \frac{\partial J}{\partial Y_{c,\mathbf{n}}} \frac{X_{c,\mathbf{n}} - \mu_c}{\gamma_c + \epsilon} \tag{1.3.21}$$

Batch normalization layers are typically placed *between* convolutional layers and non-linearities. I.e. consider the input $X = WU$ for some linear operation on the previous layer's activations U with weights W.

We see that it has the particular benefit of removing the sensitivity of our network initial weight scale, as on the forward pass $BN(aWU) = BN(WU)$. It is also particularly useful for backpropagation, as an increase in weights leads to *smaller* gradients [39], making the network far more resilient to the problems of vanishing and exploding gradients:

$$\begin{aligned}\frac{\partial BN((aW)U)}{\partial U} &= \frac{\partial BN(WU)}{\partial u} \\ \frac{\partial BN((aW)U)}{\partial (aW)} &= \frac{1}{a}\cdot\frac{\partial BN(WU)}{\partial W}\end{aligned} \tag{1.3.22}$$

1.4 Relevant Architectures

In this section we briefly review some relevant CNN architectures that will be helpful to refer back to in this thesis.

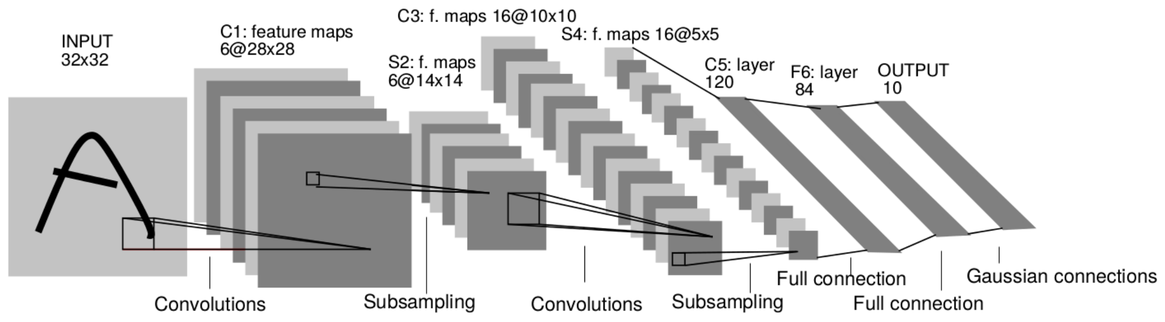

Figure 1.9: **LeNet-5 architecture.** The 'original' CNN architecture used for handwriting recognition. LeNet has 2 convolutional and 3 fully connected layers making 5 parameterized layers. After the second convolutional layer, the $16 \times 5 \times 5$ pixel output is unravelled to a 400 long vector. Image taken from **lecun_gradient-based_1998.**

1.4.1 LeNet

LeNet-5 [22] is a good network to start with: it is simple yet contains many of the layers used in modern CNNs. Shown in Figure 1.9 it has two convolutional and three fully connected layers. The outputs of the convolutional layers are passed through a sigmoid nonlinearity and downsampled with average pooling. The first two fully-connected layers also have sigmoid nonlinearities. The loss function used is a combination of tanh fucntions and MSE loss.

1.4.2 AlexNet

AlexNet [25] shown in Figure 1.10 is arguably one of the most important architectures in the development in CNNs as it was able to experimentally prove that CNNs can be used for complex tasks. This required many innovations. In particular, they used multiple GPUs to do fast processing on large images, used the ReLU to avoid saturation, and added dropout to aid generalization. Training of AlexNet on 2 GPUs available in 2012 takes roughly a week.

The first layer uses convolutions with a spatial support of 11×11, followed by 5×5 and 3×3 for the final three layers.

1.4.3 VGGnet

The Visual Geometry Group (VGG) at Oxford came second in the ILSVRC challenge in 2014 with their VGG-nets [30], but remains an important network for some of the design choices it inspired. In particular, their optimal network was much deeper than AlexNet, with 19 convolutional layers on top of each other before 3 fully connected layers. These convolutional layers all used the smaller 3×3 seen only at the back of AlexNet.

This network is particularly attractive due its simplicity, compared to the more complex Inception Network [40] which won the 2014 ILSVRC challenge. VGG-16, the 16 layer variant of VGG stacks two or three convolutional layers (and ReLUs) on top of each other

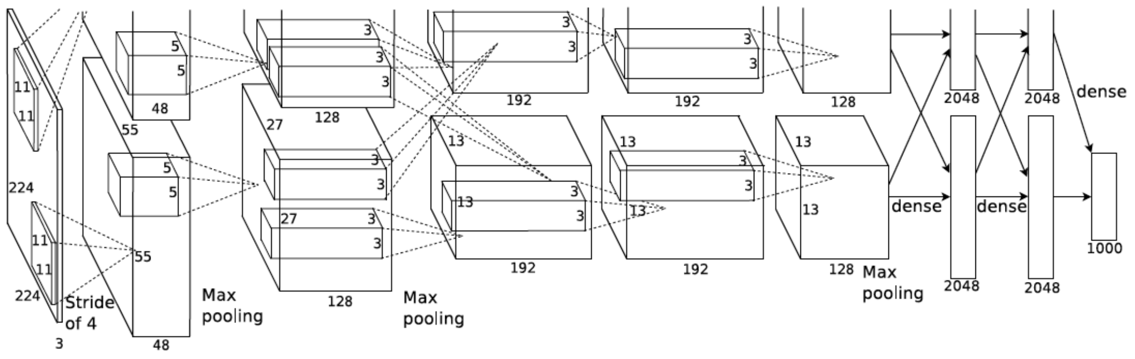

Figure 1.10: **The AlexNet architecture.** Designed for the ImageNet challenge, AlexNet may look like Figure 1.9 but is much larger. Composed of 5 convolutional layers and 3 fully connected layers. Figure taken from [25].

before reducing spatial size with max pooling. After processing at five scales, the resulting $512 \times 14 \times 14$ activation is unravelled and passed through a fully connected layer.

These VGG networks also marked the start of a trend that has since become common, where channel depth is doubled after pooling layers. The doubling of channels and quartering the spatial size still causes a net reduction in the number of activations.

1.4.4 The All Convolutional Network

The All Convolutional Network [32] introduced two popular modifications to the VGG networks:

- They argued for the removal of max pooling layers, saying that a 3×3 convolutional layer with stride 2 works just as well.
- They removed the fully connected layers at the end of the network, replacing them with 1×1 convolutions. Note that a 1×1 convolution still has shared weights across all spatial locations. The output layer then has size $C_L \times H \times W$, where H, W are many times smaller than the input image size, and the vector of C_L coefficients at each spatial location can be interpreted as a vector of scores marking the presence/absence of C_L different shapes. For classification, the output can be averaged over all spatial locations, whereas for localization it may be useful to keep this spatial information.

The new network was able to achieve

1.4.5 Residual Networks

Resiudal Networks or ResNets won the 2015 ILSVRC challenge, introducing the residual layer. Most state of the art models today use this residual mapping in some way [34], [35].

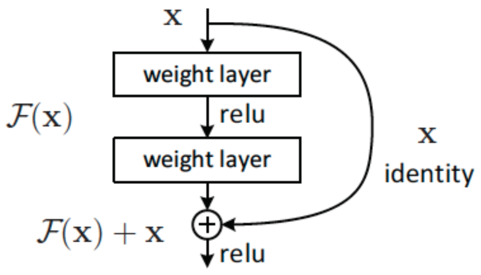
x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 1.11: **The residual unit from ResNet.** A residual unit. The identity mapping is always present, and the network learns the difference from the identity mapping, $\mathcal{F}(x)$. Taken from [33].

The inspiration for the residual layer came came from the difficulties experienced in training deeper networks. Often, adding an extra layer would *decrease* network performance. This is counter-intuitive as the deeper layers could simply learn the identity mapping, and achieve the same performance.

To promote the chance of learning the identity mapping, they define a residual unit, shown in Figure 1.11. If a desired mapping is denoted $\mathcal{H}(x)$, instead of trying to learn this, they instead learn $\mathcal{F}(x) = \mathcal{H}(x) - x$. Doing this promotes a strong diagonal in the Jacobian matrix which improve conditioning for gradient descent.

Recent analysis of a ResNet without nonlinearities [41], [42] proves that SGD fails to converge for deep networks when the network mapping is far away from the identity, suggesting that a residual mapping is a good thing to do.

References

[1] S. Boyd and L. Vandenberghe, *Convex Optimization.* New York, NY, USA: Cambridge University Press, 2004.

[2] D. R. Wilson and T. R. Martinez, “The general inefficiency of batch training for gradient descent learning”, eng, *Neural Networks: The Official Journal of the International Neural Network Society*, vol. 16, no. 10, pp. 1429–1451, Dec. 2003.

[3] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le, “Don’t Decay the Learning Rate, Increase the Batch Size”, *arXiv:1711.00489 [cs, stat]*, Nov. 2017. arXiv: 1711.00489 `[cs, stat]`.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016.

[5] L. Bottou, “Stochastic Gradient Descent Tricks”, en-US, vol. 7700, Jan. 2012.

[6] G. Montavon, G. Orr, and K.-R. Mller, *Neural Networks: Tricks of the Trade*, 2nd. Springer Publishing Company, Incorporated, 2012.

[7] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, “Efficient BackProp”, en, in *Neural Networks: Tricks of the Trade*, ser. Lecture Notes in Computer Science 7700, G. Montavon, G. B. Orr, and K.-R. Müller, Eds., Springer Berlin Heidelberg, 2012, pp. 9–48.

[8] Y. A. Ioannou, “Structural Priors in Deep Neural Networks”, PhD thesis, Department of Engineering, University of Cambridge, Trumpington Street, Cambridge CB2 1PZ, United Kingdom, Sep. 2017.

[9] D. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization”, *arXiv:1412.6980 [cs]*, Dec. 2014. arXiv: 1412.6980 `[cs]`.

[10] J. Duchi, E. Hazan, and Y. Singer, “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”, *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[11] M. D. Zeiler, “ADADELTA: An Adaptive Learning Rate Method”, *arXiv:1212.5701 [cs]*, Dec. 2012. arXiv: 1212.5701 `[cs]`.

[12] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain [J]”, *Psychol. Review*, vol. 65, pp. 386–408, Dec. 1958.

[13] X. Glorot, A. Bordes, and Y. Bengio, "Deep Sparse Rectifier Neural Networks", in *International Conference on Artificial Intelligence and Statistics*, 2011, pp. 315–323.

[14] V. Nair and G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines", in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010, pp. 807–814.

[15] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks", in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[16] P. Lennie, "The cost of cortical computation", eng, *Current biology: CB*, vol. 13, no. 6, pp. 493–497, Mar. 2003.

[17] M. L. Minsky and S. A. Papert, *Perceptrons: Expanded Edition.* Cambridge, MA, USA: MIT Press, 1988.

[18] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators", *Neural Networks*, vol. 2, no. 5, pp. 359–366, Jan. 1989.

[19] G. Cybenko, "Approximation by superpositions of a sigmoidal function", en, *Mathematics of Control, Signals and Systems*, vol. 2, no. 4, pp. 303–314, Dec. 1989.

[20] B. Widrow and M. E. Hoff, "Neurocomputing: Foundations of Research", in, J. A. Anderson and E. Rosenfeld, Eds., Cambridge, MA, USA: MIT Press, 1988, pp. 123–134.

[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1", in, D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, Eds., Cambridge, MA, USA: MIT Press, 1986, pp. 318–362.

[22] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[23] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex", eng, *The Journal of Physiology*, vol. 160, pp. 106–154, Jan. 1962.

[24] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints", *International Journal of Computer Vision*, vol. 60, pp. 91–110, 2004.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", in *NIPS*, Curran Associates, Inc., 2012, pp. 1097–1105.

[26] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge", *arXiv:1409.0575 [cs]*, Sep. 2014. arXiv: 1409.0575 `[cs]`.

[27] C. Cortes and V. Vapnik, “Support-vector networks”, en, *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep. 1995.

[28] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in PyTorch”, Oct. 2017.

[29] D. Mishkin, N. Sergievskiy, and J. Matas, “Systematic evaluation of CNN advances on the ImageNet”, *arXiv:1606.02228 [cs]*, Jun. 2016. arXiv: 1606.02228 `[cs]`.

[30] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition”, *arXiv:1409.1556 [cs]*, Sep. 2014. arXiv: 1409.1556 `[cs]`.

[31] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten, “Densely Connected Convolutional Networks”, in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul. 2017, pp. 2261–2269. arXiv: 1608.06993.

[32] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, “Striving for Simplicity: The All Convolutional Net”, *arXiv:1412.6806 [cs]*, Dec. 2014. arXiv: 1412.6806 `[cs]`.

[33] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition”, in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp. 770–778. arXiv: 1512.03385.

[34] S. Xie, R. B. Girshick, P. Dollár, Z. Tu, and K. He, “Aggregated Residual Transformations for Deep Neural Networks”, *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5987–5995, 2017.

[35] S. Zagoruyko and N. Komodakis, “Wide Residual Networks”, en, May 2016.

[36] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors”, *arXiv:1207.0580 [cs]*, Jul. 2012. arXiv: 1207.0580 `[cs]`.

[37] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[38] Y. Gal and Z. Ghahramani, “Dropout As a Bayesian Approximation: Representing Model Uncertainty in Deep Learning”, in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML’16, JMLR.org, 2016, pp. 1050–1059.

[39] S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, *arXiv:1502.03167 [cs]*, Feb. 2015. arXiv: 1502.03167 `[cs]`.

[40] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going Deeper With Convolutions”, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

[41] P. L. Bartlett, S. N. Evans, and P. M. Long, "Representing smooth functions as compositions of near-identity functions with implications for deep network optimization", *arXiv:1804.05012 [cs, math, stat]*, Apr. 2018. arXiv: 1804.05012 `[cs, math, stat]`.

[42] P. L. Bartlett, D. P. Helmbold, and P. M. Long, "Gradient descent with identity initialization efficiently learns positive definite linear transformations by deep residual networks", *arXiv:1802.06093 [cs, math, stat]*, Feb. 2018. arXiv: 1802.06093 `[cs, math, stat]`.