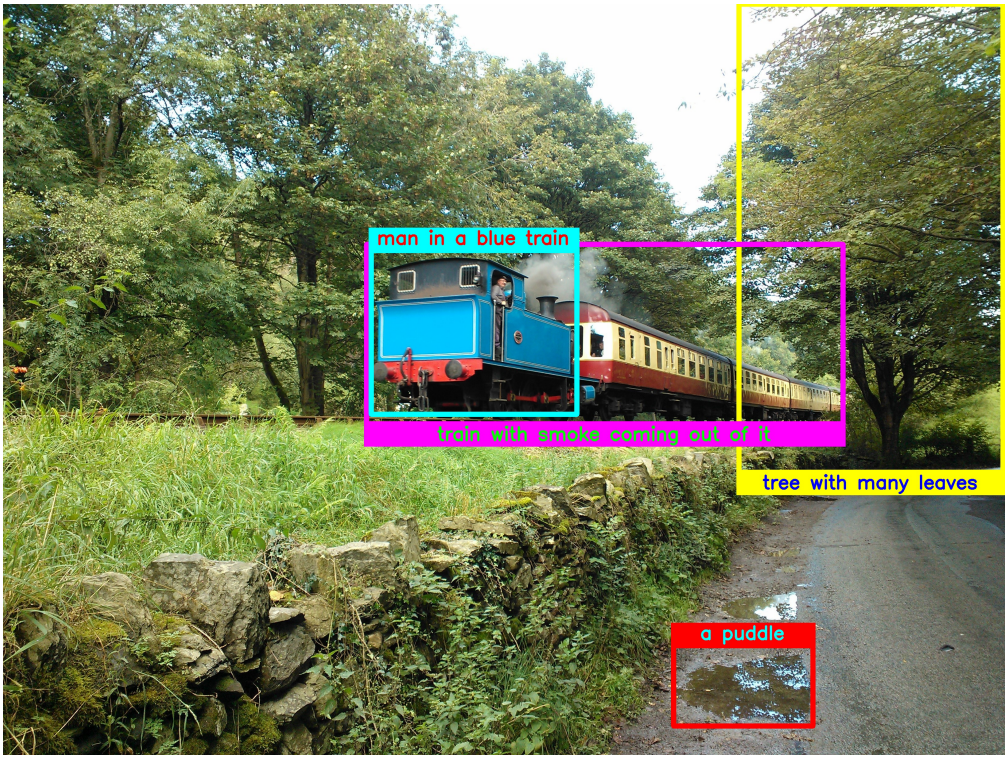
University of Cambridge
Engineering Part IIB
Module 4F12: Computer Vision

Handout 5: Deep Learning for Computer Vision

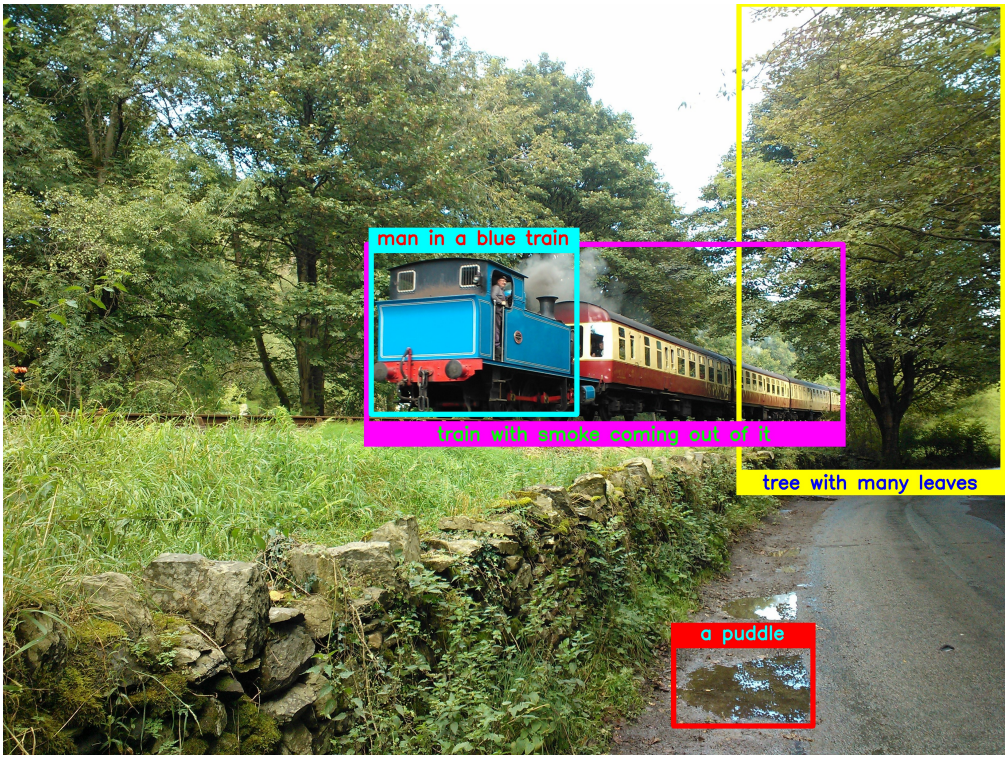


Matthew Johnson
November 2023

Artificial Intelligence

Complex, interconnected networks of neurons, like the human brain, are the only known way of achieving general intelligence. Because of this, many efforts in the area of Artificial Intelligence attempt to recreate the different structures of the brain. Systems called *deep neural nets* (DNNs) have proven helpful for more targeted AI tasks, like image understanding:

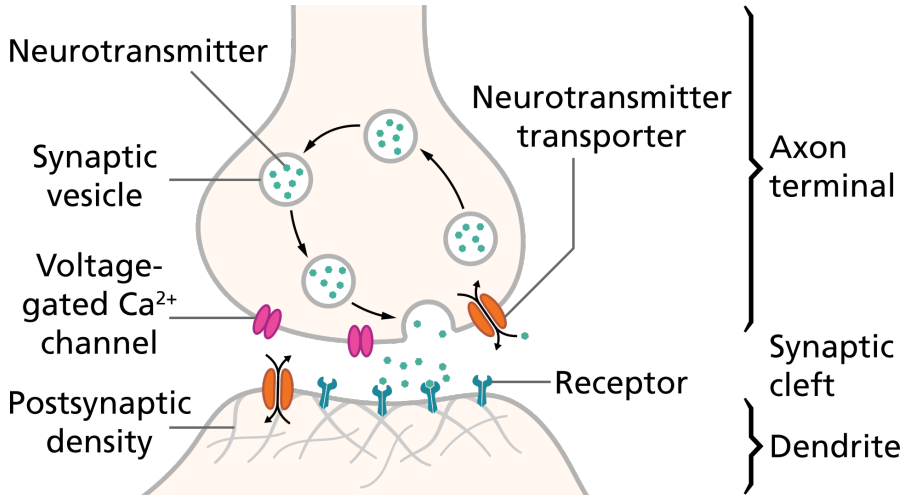
`https://aka.ms/densecaptions`



DNNs can seem intelligent because they do things we usually associate with human intelligence, like scene understanding and object recognition. In reality, they are highly specialised, trained from vast corpora of data, and constructed from elements that mimic biological neural networks in useful yet incomplete ways. We will explore the underlying mathematics behind DNNs, analyse their structures and design principles, and study state-of-the-art architectures and techniques.

Neurons

Before we dive into the maths, let us perform an (extremely brief) overview of neurons and how they work. Below is a diagram of a synapse:



The ways in which neurons work together involve complex interactions between electrical signals and neurochemistry that are very much outside the scope of this lecture. That said, for our purposes it is important to understand three key concepts:

- A neuron has both dendrites (inputs) and axons (outputs) which connect it to other neurons
- A neuron will produce an exciting or inhibiting signal along its axons based upon activations from its dendrites in a non-linear way
- Each dendrite contributes to whether a signal is produced in different proportions, which change over time

We will focus on these three principles and how to use them to build a simple mathematical model of a neuron.

Perceptrons

The perceptron algorithm was invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt [14]. It was designed to be implemented in hardware for the purpose of image recognition, and was the marvel of the age. The New York Times reported the perceptron to be "the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence." [12].

Figure 1: Frank Rosenblatt

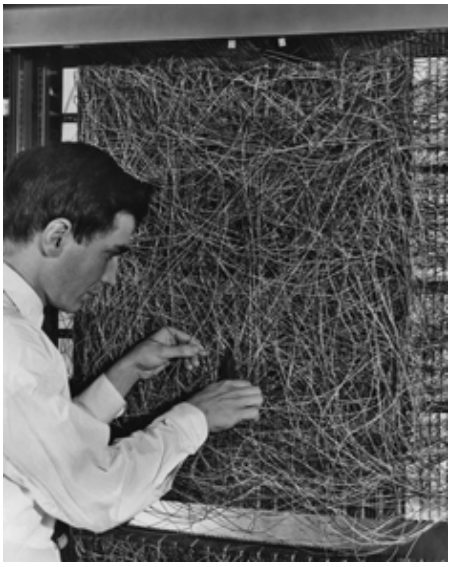


Figure 2: The Mark 1 perceptron

Rosenblatt based his designs upon earlier theoretical work by Warren McCulloch and Walter Pitts in 1943 on the Threshold Logic Unit [11], arguably the first artificial neuron. The idea in both cases was to create a simple mathematical model of a neuron. In the case of the perceptron, this model was then expressed in hardware. The entire machine represented a single neuron, with 400 inputs taken from a 20x20 image sense to create a binary output.

Perceptrons, cont.

Given an input $\mathbf{x}$, the mathematical model for the output y of a perceptron is calculated as:

$$y(\mathbf{x}) = f(\mathbf{w}^{\mathrm{T}}\mathbf{x})$$

where $\mathbf{w}$ are the weights on the input vector $\mathbf{x}$ and f is a step function of the form:

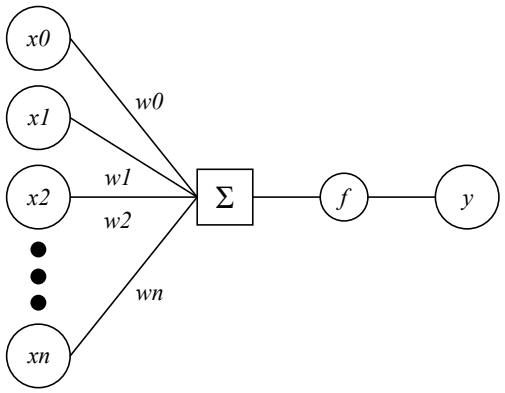


Figure 3: A perceptron. There is one weight per input, and a single output.

$$f(x_i) = \begin{cases} +1 & x_i \geq 0 \\ -1 & x_i < 0 \end{cases}$$

The fundamental problem at hand is how to learn the values of $\mathbf{w}$. For this, we use the *perceptron criterion*:

$$E_{\mathrm{P}}(\mathbf{w}) = -\sum_{n \in \mathcal{M}} \mathbf{w}^{\mathrm{T}}\mathbf{x}_n t_n$$

where $t_n \in \{-1, +1\}$ for negative and positive examples and $\mathcal{M}$ is the set of misclassified examples.

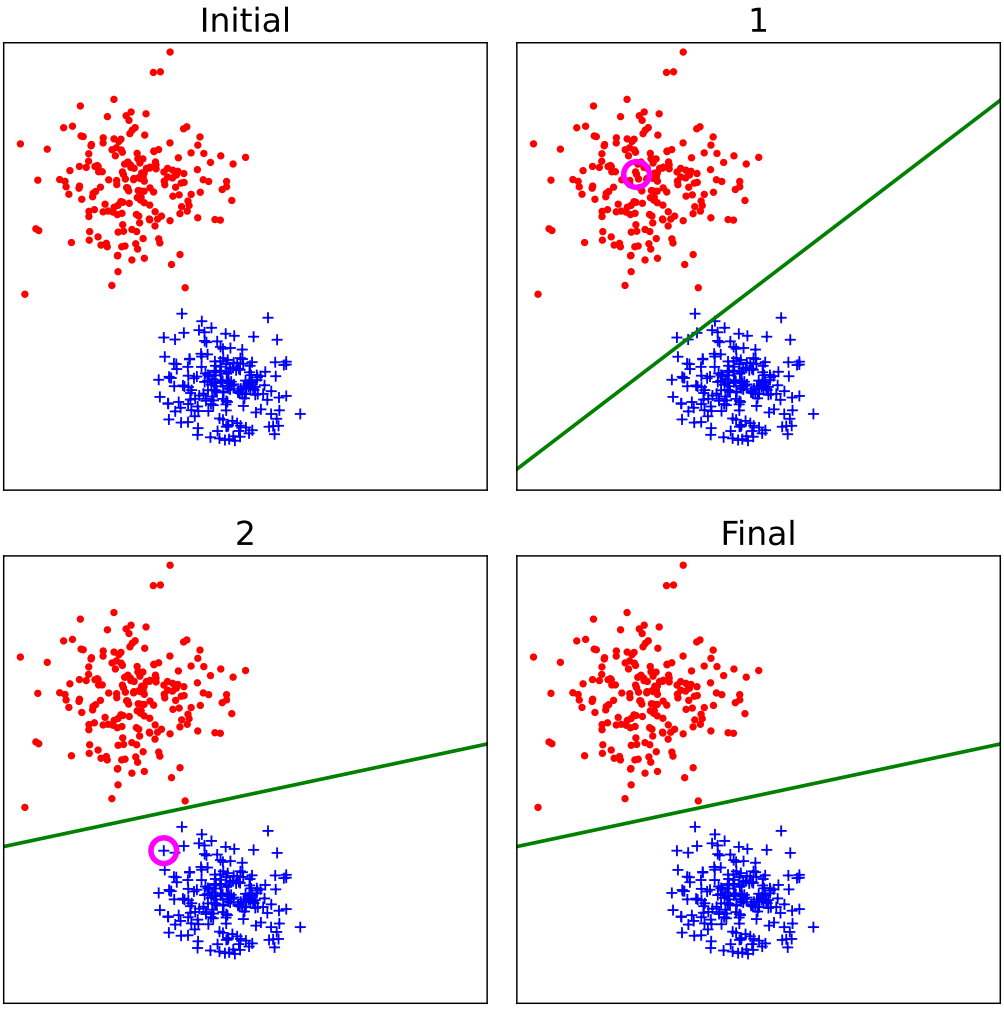
We can use the gradient of this function to update the weights on a per-example basis:

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta \nabla E_{\mathrm{P}}(\mathbf{w}) = \mathbf{w}^{\tau} + \eta \mathbf{x}_n t_n$$

where η is the learning rate parameter and τ is an integer that indexes the steps of the algorithm.

Binary Classification

Perceptrons can be used to predict the membership between two classes. Take, for example, the two sets of points below. We can train a perceptron to separate these two classes:



The perceptron only needs to be shown two examples to find a separating decision boundary.

Combining Perceptrons

As originally proposed, perceptrons can only answer yes/no questions. However, we can combine perceptrons to tackle multi-class problems using *1 versus all*. Instead of a single vector, each perceptron is now a column in a matrix:

$$\mathbf{y} = f(\mathbf{W}\mathbf{x})$$

where $\mathbf{W}$ are the weights on the input vector $\mathbf{x}$ and f is the same step function, applied on a per-index basis on the output vector:

$$f(x_i) = \begin{cases} +1 & x_i \geq 0 \\ -1 & x_i < 0 \end{cases}$$

Each perceptron focuses on a single question: is this input a member of my class, or not? We can then look at multiple outputs and choose the class that corresponds to the classifier that has a positive result. The maths stay much the same, though we are going to change the formulation slightly. Instead of including the bias as an extra dimension in the data, we are going to use a common convention in modern neural net architectures and separate it out as a separate *bias* term, $\mathbf{b}$, as follows:

$$\mathbf{y} = f(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Activation Functions

Compute the y_0 and y_1 given the following inputs:

$$\mathbf{x} = \begin{bmatrix} 0.3 \\ -0.5 \end{bmatrix}$$
$$\mathbf{W} = \begin{bmatrix} 0.55 & -0.12 \\ -0.32 & 0.5 \end{bmatrix}$$
$$\mathbf{b} = \begin{bmatrix} 0.4 \\ -0.2 \end{bmatrix}$$

$$h_0 = 0.55 \times 0.3 + -0.12 \times -0.5 + 0.4 = 0.625$$
$$h_1 = -0.32 \times 0.3 + 0.5 \times -0.5 - 0.2 = -0.546$$
$$y_0 = 1$$
$$y_1 = -1$$

Try the exercise again with $\mathbf{x} = \begin{bmatrix} -0.4 \\ 1 \end{bmatrix}$:

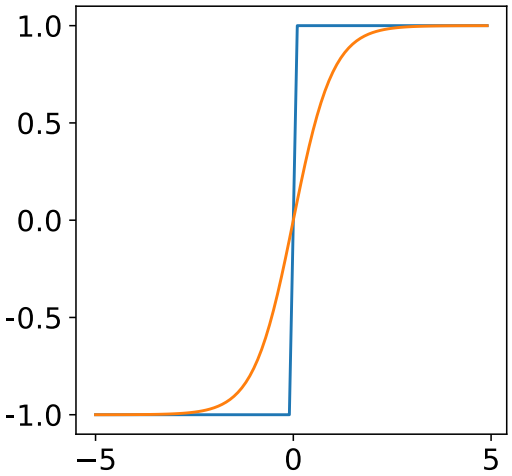
$$h_0 = 0.55 \times -0.4 + -0.12 \times 1 + 0.4 = 0.06$$
$$h_1 = -0.32 \times -0.4 + 0.5 \times 1 - 0.2 = 0.428$$
$$y_0 = 1$$
$$y_1 = 1$$

Can you see the issue?

Activation Functions, cont.

As you discovered in these two examples, the step function the perceptron uses cannot distinguish between different positive results, which is a requirement for classifying more than two classes. Thankfully, it is just one example of an *activation function*, and there are many others. One which is a close approximation to the step function is the hyperbolic tangent:

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



we now have a new formulation for the perceptron:

$$\mathbf{y} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b})$$

This new formulation, which we will call a *multiperceptron*, has several advantages. The hyperbolic tangent, in addition to being roughly linear from -1 to 1, saturates to 1 or -1 as well, thus providing a useful *non-linearity*, which was the driving force behind the use of the step function in the original perceptron. It also creates a function that is fully differentiable, which means we can learn the values of $\mathbf{W}$ via *gradient descent*.

Gradient Descent

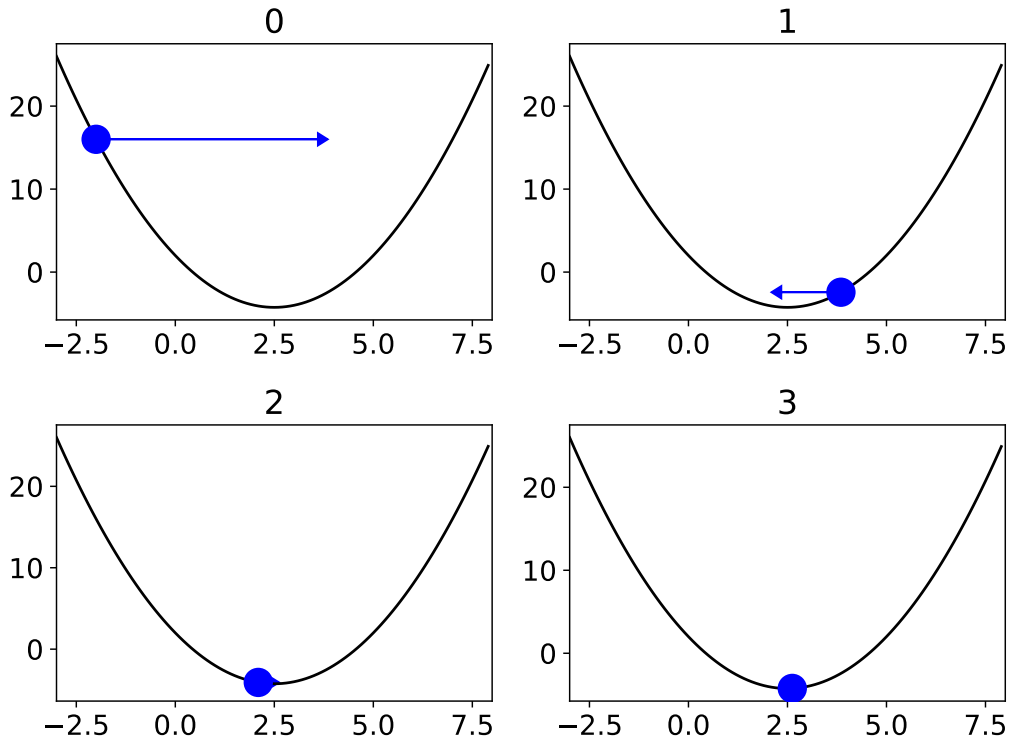
As you may recall, gradient descent is the practice of using the gradient of a function to find the input values that correspond to its minimum. Take for example the parabola defined by:

$$y(x) = x^2 - 5x + 2$$

This can be thought of as an *objective function* of x. We can use its gradient:

$$y'(x) = 2x - 5$$

to find the value of x that minimises $y(x)$. A step size or *learning rate* η is used to control how far we move in the direction of the gradient.



Backpropagation

The use of gradient descent to discover the parameters of a network of perceptrons is called *backpropagation*, and was discovered by Paul J. Werbos, who described it in his 1974 thesis *Beyond regression: New tools for prediction and analysis in the behavioural sciences*[16]. In order to use backpropagation here we need two things: a (differentiable) way of producing a desired output value, and an (also differentiable) objective function for scoring that value.

The task we are currently interested in performing is *classification*, where we label an input as being a member of one of several classes C. One way of achieving this is by generating a probability distribution $P(c \mid \mathbf{x})$, which defines a distribution over C such that $\sum_{c \in C} P(c \mid \mathbf{x}) = 1$ and where the magnitude of $P(c \mid \mathbf{x})$ corresponds to the classifier's confidence that $\mathbf{x}$ belongs to class c. A typical choice to produce $P(c \mid \mathbf{x})$ for a multiperceptron is the Softmax function s:

$$
\begin{aligned}
P(c = i \mid \mathbf{x}) &= s(y_i) \\
s(y_i) &= \frac{e^{y_i}}{\sum_j e^{y_j}} \\
\mathbf{y} &= \tanh(\mathbf{W}\mathbf{x} + \mathbf{b})
\end{aligned}
$$

Cross Entropy

Now that we have a way of generating an output ($P(c \mid \mathbf{x})$) we need a way of scoring it, *i.e.* telling whether it is a good or bad output. For this we can use the *cross entropy* function:

$$H(P, Q) = -\sum_c P(c) \log Q(c)$$

where P is the target distribution and Q is $P(c \mid \mathbf{x})$.

The cross entropy consists of two components:

$$H(P, Q) = H(P) + D_{\mathrm{KL}}(P \mid\mid Q)$$

for discrete distributions P and Q, where H is the Shannon entropy:

$$H(P) = -\sum_c P(c) \log P(c)$$

Entropy measures the *information content* of a distribution, or how much we learn from sampling. The Kullback-Leibler divergence D_{KL} measures how much one distribution diverges from another:

$$D_{\mathrm{KL}}(P \mid\mid Q) = \sum_c P(c) \log \frac{P(c)}{Q(c)}$$

The intuition here is that if P and Q are the same, then the term $\log \frac{P(c)}{Q(c)}$ will be zero and thus the divergence is also zero.

Optimisation

We can now write the objective function for a multiperceptron:

$$L(D, \mathbf{W}, \mathbf{b}) = \frac{1}{|D|} \sum_{d \in D} H(\mathbf{t}_d, P(c \mid \mathbf{x}_d))$$
$$P(c \mid \mathbf{x}_d) = s(\mathbf{W}\mathbf{x}_d + \mathbf{b})$$

where D is our dataset and $\mathbf{t}_i$ is a *one-hot* vector corresponding to the correct label for that data point. In words, we want to minimise the expected cross entropy between the output and the target distribution. Now that we have an objective, we can compute gradients for $\mathbf{W}$ and $\mathbf{b}$ and optimize them directly.

At first glance the task of computing partial derivatives of L with respect to $\mathbf{W}$ and $\mathbf{b}$ seems daunting. However, we have chosen our functions very carefully. Using the chain rule, we first look at partial derivatives with respect to the softmax function, but immediately run into a problem, in that y_i is in both the numerator and denominator of s. As such we need to index the derivative:

$$\frac{\partial s_i}{\partial y_i} = s_i(1 - s_i)$$
$$\frac{\partial s_j}{\partial y_i} = -s_i s_j, \; j \neq i$$

Optimisation, cont.

Breaking things out in this way, we can now look at our first partial derivatives for L (for a single data point):

$$
\begin{aligned}
L &= H(\mathbf{t}, P(c \mid \mathbf{x})) \\
L &= -\sum_i t_i \log s_i \\
\frac{\partial L}{\partial s_i} &= -\sum_k t_k \frac{\partial \log s_k}{\partial y_k} \\
&= -\sum_k t_k \frac{1}{s_k} \frac{\partial s_k}{\partial y_k} \\
&= -t_i(1 - s_i) - \sum_{k \neq i} t_k \frac{1}{s_k}(-s_k s_i) \\
&= -t_i + t_i s_i + \sum_{k \neq i} t_k s_i \\
&= s_i \left(\sum_k t_k \right) - t_i
\end{aligned}
$$

Remember that $\mathbf{t}$ is a one-hot vector, and so we get the final result:

$$\frac{\partial L}{\partial s_i} = s_i - t_i$$

Optimisation, cont.

Continuing with the chain rule, we can now look at the partial derivatives for the hyperbolic tangent:

$$\begin{aligned} y_i &= \tanh(h_i) \\ h_i &= \mathbf{W}\mathbf{x}_i + \mathbf{b} \\ \frac{\partial y_i}{\partial h_i} &= 1 - y_i^2 \end{aligned}$$

As we are performing matrix calculus, is it helpful to think about vectors of partial derivatives moving forward, which we will denote $\delta = \left\{ \frac{\partial L}{\partial h_i}, \ \forall i \right\}$. Using δ we can compute the partial derivatives for the weights and biases:

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{W}} &= \mathbf{x}\delta^{\mathrm{T}} \\ \frac{\partial L}{\partial \mathbf{b}} &= \sum_i \delta_i \end{aligned}$$

By subtracting these values from the current values of $\mathbf{W}$ and $\mathbf{b}$, we can perform gradient descent to update the weights and biases and minimise our objective function, with the aim that $P(c \mid x_d) \approx \mathbf{t}_d$ for all $d \in D$.

Batching

In the previous slides we looked at the equations for only a single data point $\mathbf{x}$ and its target $\mathbf{t}$. However, we really want to subtract the value for the full definition of L which incorporates an expectation over all data points:

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{1}{|D|} \sum_{d \in D} \mathbf{x}_d \delta_d^{\mathrm{T}}$$
$$\frac{\partial L}{\partial \mathbf{b}} = \frac{1}{|D|} \sum_{d \in D} \sum_i \delta_{di}$$

This is a problem, as we need to compute the full gradient before we can update the weights and biases. This is computationally expensive, and so we instead use a technique called *batching*, where we compute the gradient for a subset of the data points, and then update the weights and biases.

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{1}{|D_B|} \sum_{d \in D_B} \mathbf{x}_d \delta_d^{\mathrm{T}}$$
$$\frac{\partial L}{\partial \mathbf{b}} = \frac{1}{|D_B|} \sum_{d \in D_B} \sum_i \delta_{di}$$
$$D_B \subseteq D$$

This is repeated until we have processed all the data points in the dataset. If these subsets are chosen at random, this is called *stochastic gradient descent*.

Datasets

At this point, we will briefly discuss the correct usage of data. Data should be divided into three sets:

1. **Training** – Fitting the model
2. **Validation** – Determining hyperparameters
3. **Test** – Held out for evaluation.

The two commandments of data science can be written in terms of these three sets:

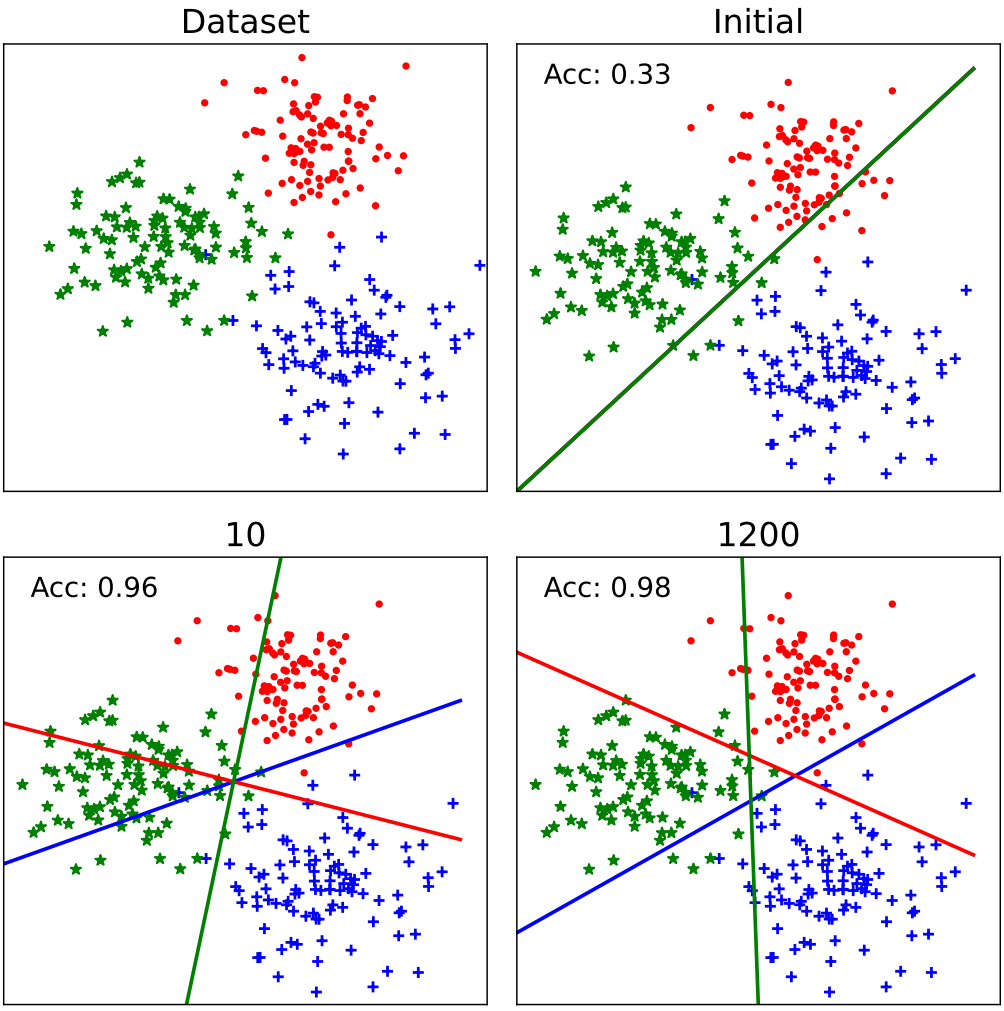
1. Thou shalt not train on the test data
2. Thou shalt not determine hyperparameters on test data

These both derive from the central goal of machine learning: *generalisation*. When we train a model, we want to ensure that it can make reasonable predictions when shown new, unseen data. This is why we take a portion of our labelled data and set it aside for use as test data. If we use the test data to train our model, we have no way of knowing if our model will generalise to that test data, and thus the results are meaningless.

The same is true for hyperparameters. If we use the test data to determine hyperparameters, we have no way of knowing if our model will generalise to that test data, and thus the results are meaningless. Thus, we set aside validation data for this purpose. However, we can safely incorporate the validation data when we train the final model as we still have the held-out test data to evaluate the model's performance.

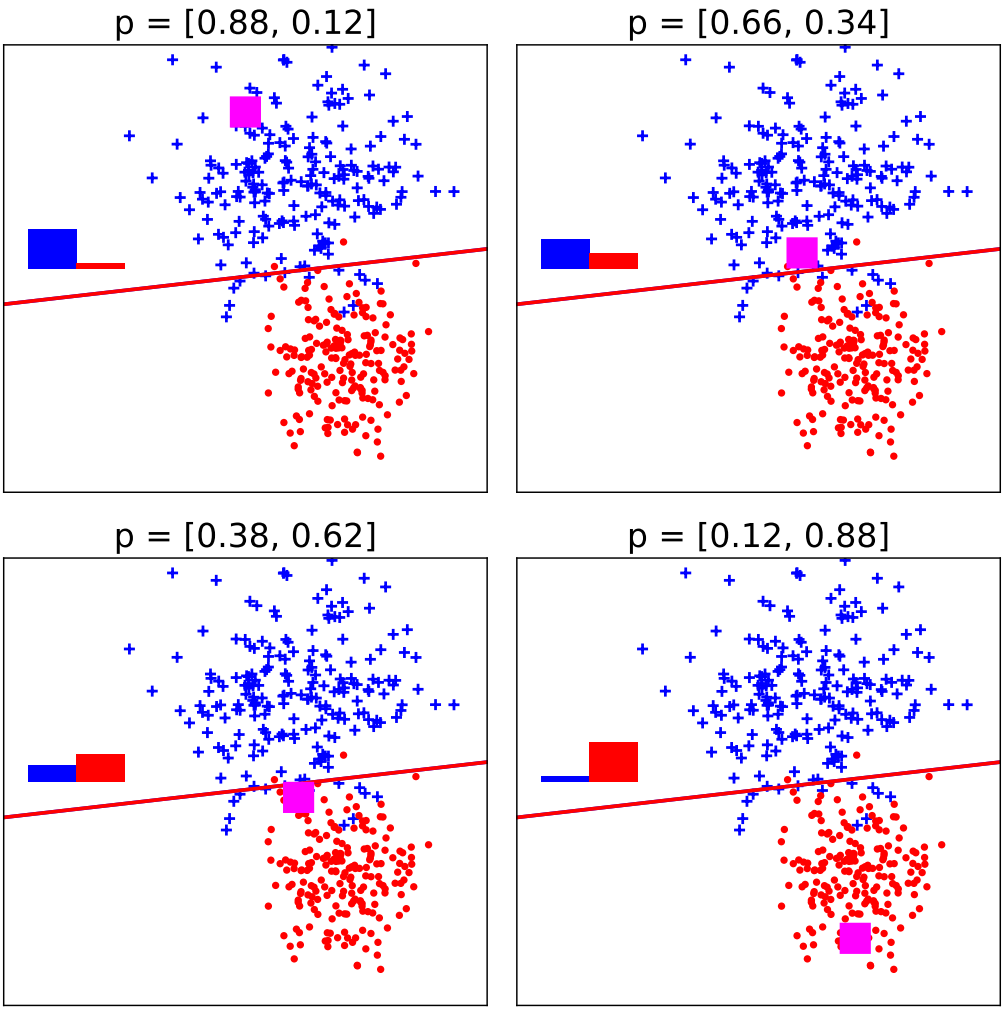
N-ary Classification

Our multiperceptron is capable of predicting labels for any number of classes, not just the two classes that a generic perceptron can handle. Below we see a dataset containing three classes. We can train our multiperceptron to classify them:



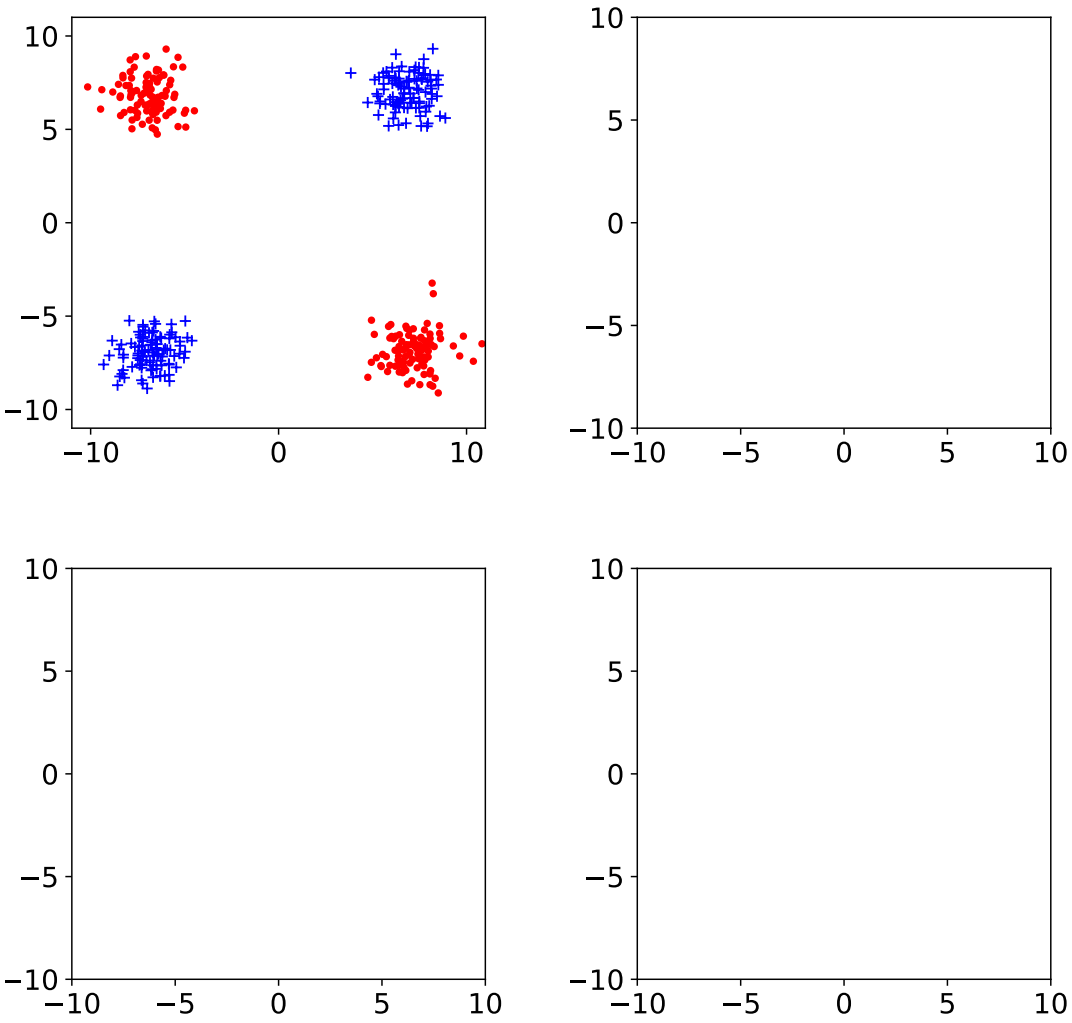
Uncertainty

Aside from the ability to handle multiple classes, the multiperceptron also provides us with a measure of uncertainty. As a point moves closer to or farther away from the decision boundary, the model becomes less or more certain. See what happens as the square moves from the top to the bottom:



The Linearity Problem

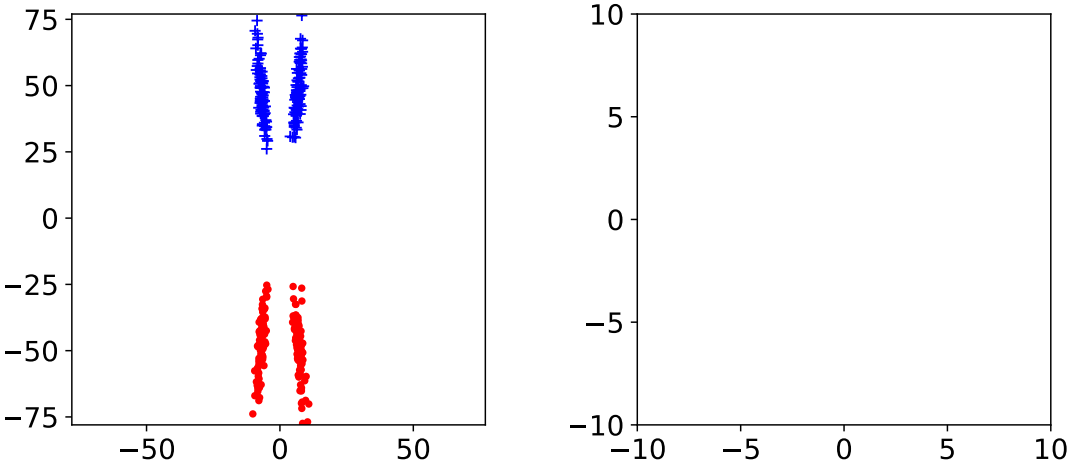
The multiperceptron is a powerful tool, but it has a major flaw: it can only be used on linearly separable data. Below you can see one example of data which cannot be modeled by the multiperceptron. Can you think of others?



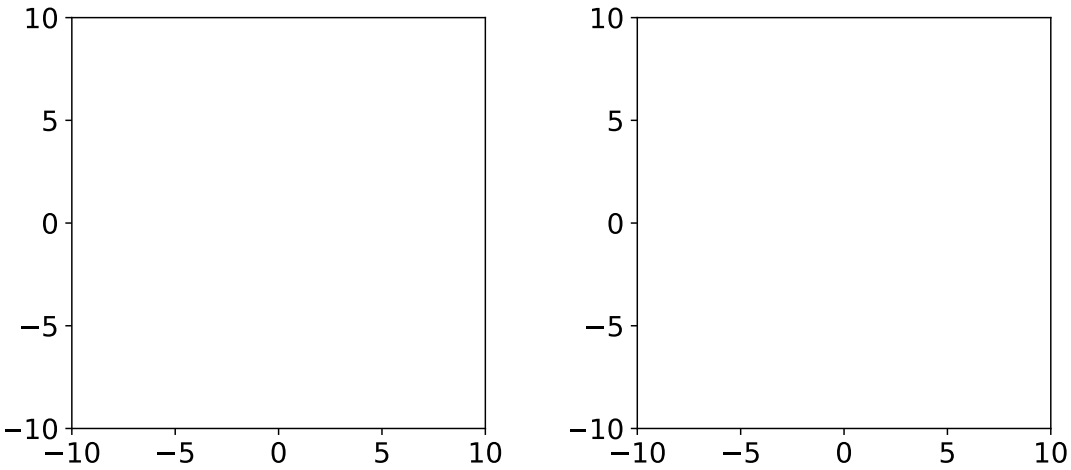
Make no mistake, there is no way to change the multiperceptron to overcome this problem. We will need to change the input data itself.

Feature Engineering

One way to overcome this problem is via *feature engineering*, whereby we engineer a feature transform that takes as input the original data and produces new data which is linearly separable and can be used as input to a multiperceptron. Take a moment to return to your examples from the previous slide and see if you can engineer a feature transform that would make the data linearly separable.



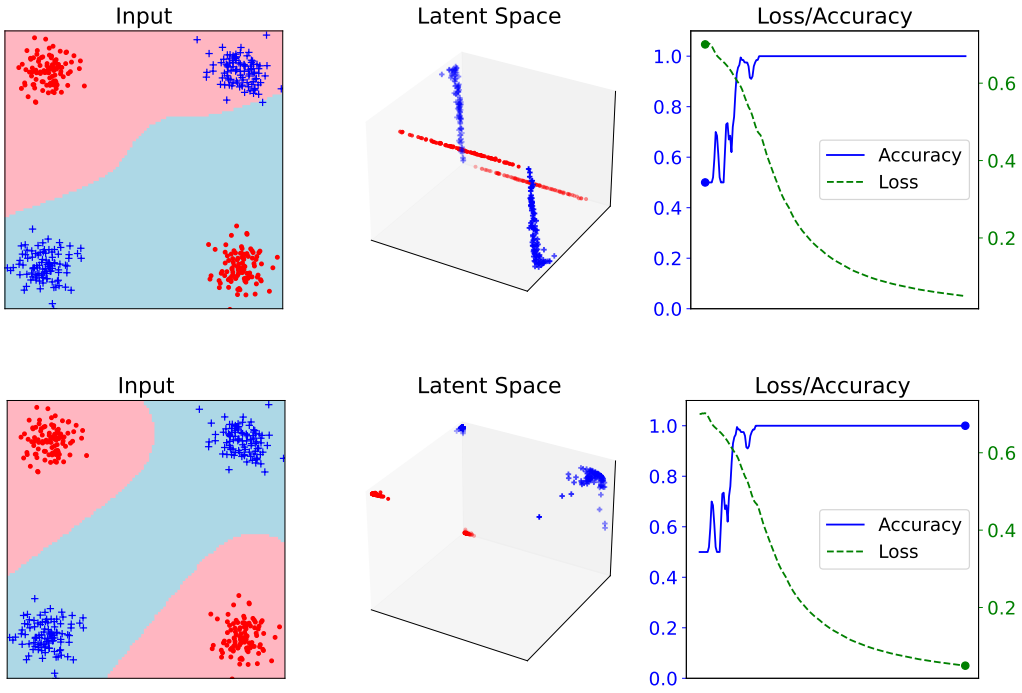
$f(x, y) = (x, xy)$



Multi-layer Perceptrons (MLPs)

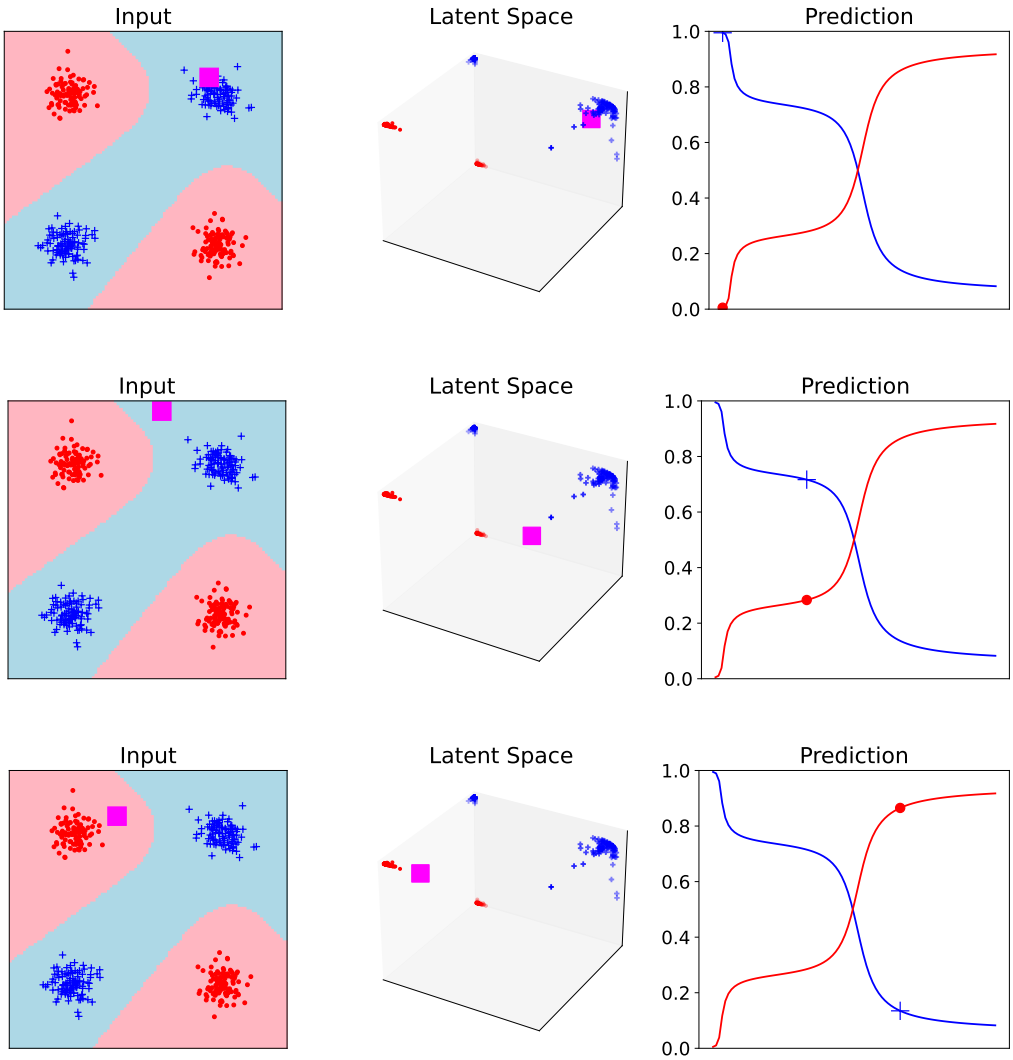
Feature engineering is an incredibly useful tool, but requires human expertise. For low-dimensional datasets like these it can be straightforward, but as data becomes more complex, engineering features becomes increasingly difficult. Ideally, we want to be able to learn the feature transform from the data itself. This is the idea behind *multi-layer perceptrons*.

We will add a multiperceptron as an intermediary step between the input data and our classifier. This multiperceptron has a different task: to learn a feature transform that makes the data linearly separable. When used in this way, we will refer to a multiperceptron as a *hidden fully-connected layer*. The output of one or more hidden layers is fed into the classifier, which is now called the *output layer*. The resulting network is called a *multi-layer perceptron*.



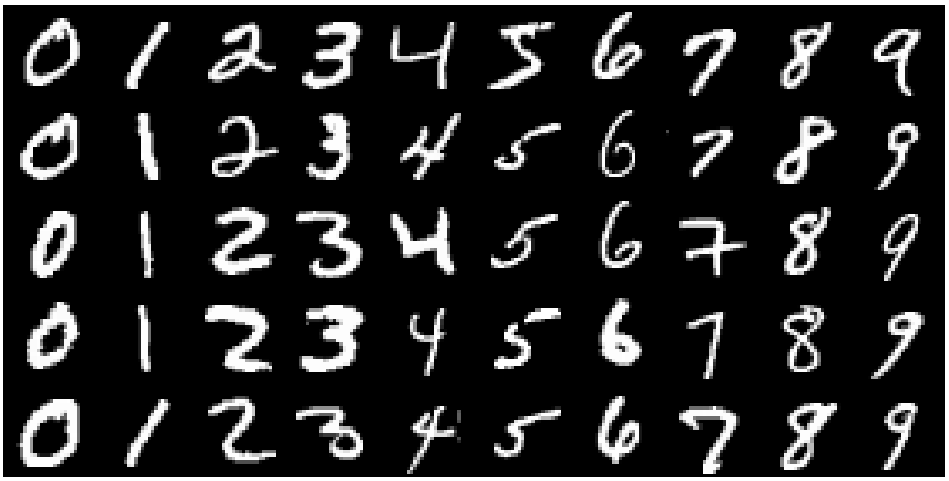
Understanding the Latent Space

The latent spaces learned by MLPs can be difficult to understand. However, one useful technique is to see how values change when we use the differential nature of the network explore how input changes cause output changes. Instead of computing partial derivatives with respect to the weights and biases of the network, we compute them with respect to the input data itself. By changing the target class and then updating the input data to minimise this new objective (while keeping the model fixed) we can get a sense for how the model makes its decisions.



Classifying Images

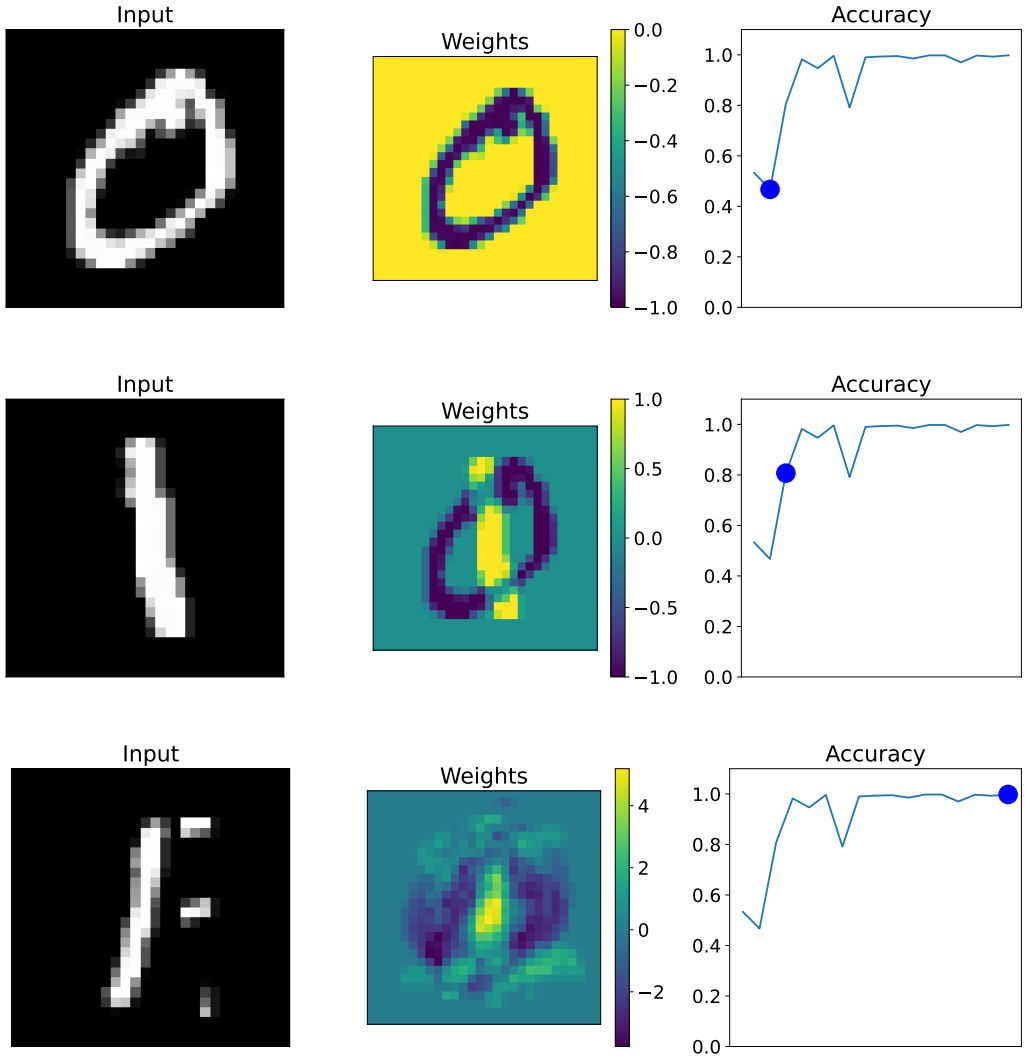
The original perceptron system took as input 20×20 images, which were flattened into a 400-dimensional vector. Let us look at a similar problem: handwritten digit classification using the MNIST dataset.



The images above were gathered from employees of the US Census Bureau and Secondary School students, consisting of approximately 250 individuals. There are 60,000 training images and 10,000 test images (sampled in a disjoint manner from the two sample groups). The images themselves are 28×28 unregistered greyscale images. Research has been performed on the dataset since the 1980s and has continued to the present day, where it is often used for debugging network architectures.

MNIST: Perceptron

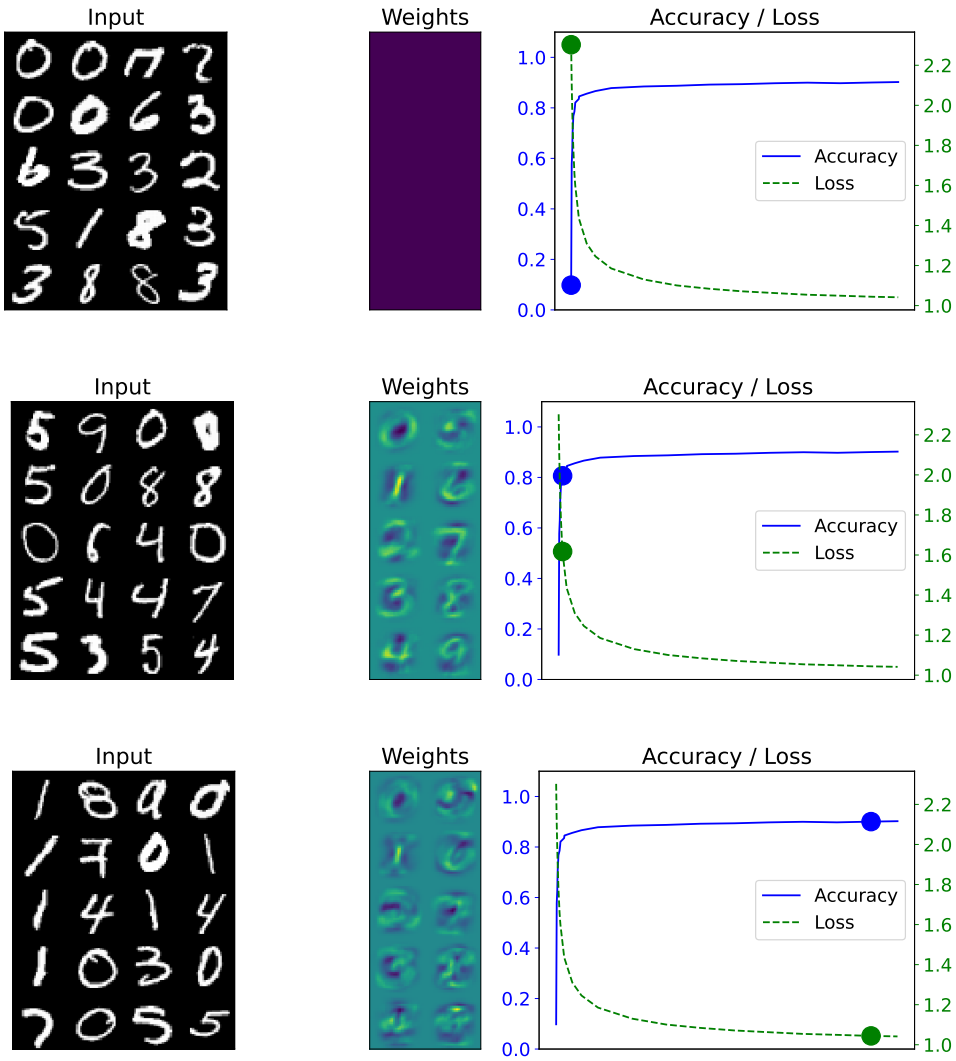
We will begin by looking at how a perceptron predicts whether an image is 1 or 0. Because the weight vector w of a perceptron is the same size as its input, we can visualize these weights as an image.



Note how the weights are being set by subtracting or adding images, so that we can see afterimages of specific 1 and 0 images in the early weights. Note also how the weight magnitude increases over time.

MNIST: Multiperceptron

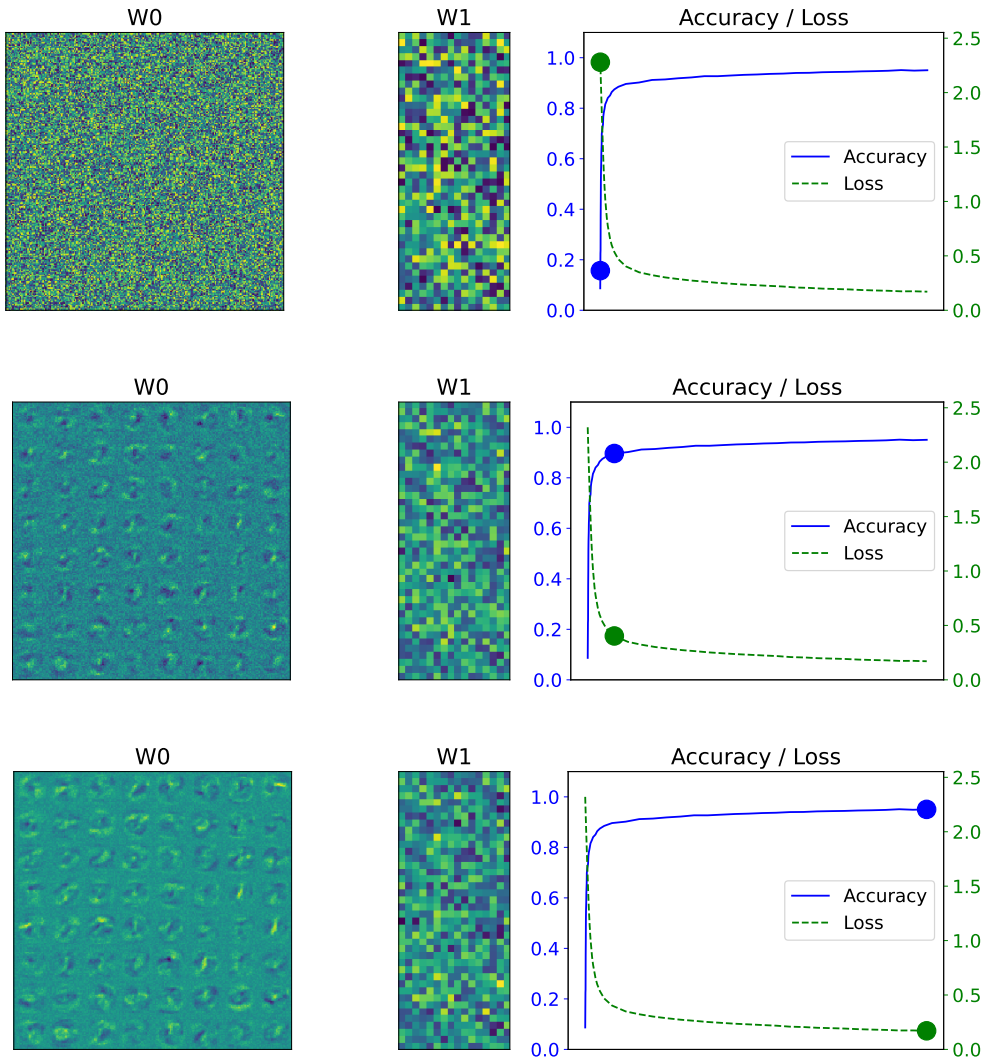
Now we will look at training a multiperceptron on a harder problem: all ten classes. We can still view the weights as images, with one weight image per row of the W matrix.



As training progresses the weights for each class become positive where that class has pixels, and negative elsewhere.

MNIST: Multi-layer Perceptron

Finally, we will look at training a multi-layer perceptron on the same problem. This network has a hidden layer dimension of 64, meaning that first the 784-dimensional image is embedded in a 64-dimensional space, and then the multiperceptron determines the classification boundary as a hyperplane in 64 dimensions. We can view the weights in the first layer the same way we have done so far, and can view the weights in the second layer as a 5×2 grid of $8x8$ images.

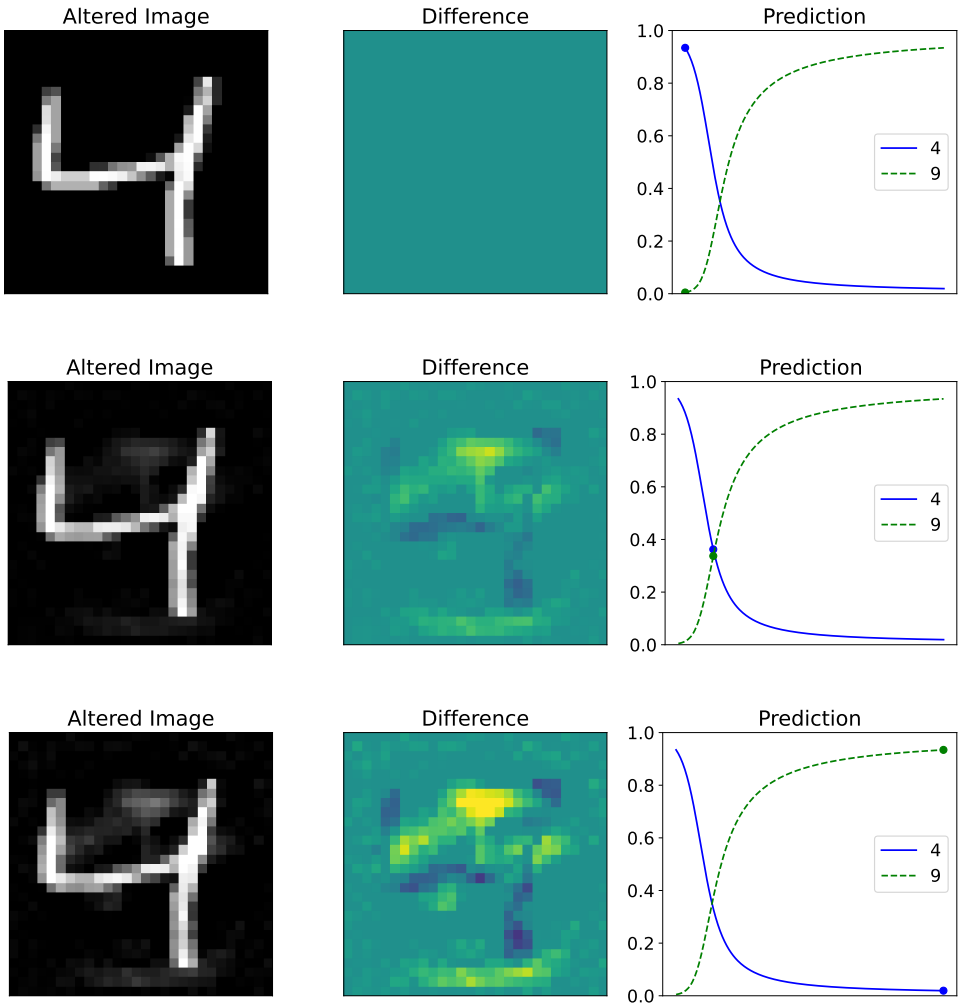


Attacking an MLP Model

As we did before, we can see how an input image needs to change to cause the network to predict a different class. However, unlike the previous example where the input and latent spaces were low dimensional, these images are embedded in a 784-dimensional space. If we alter the objective function to additionally minimise the distance in the input space:

$$L'(\mathbf{W}, \mathbf{b}) = L(d, \mathbf{W}, \mathbf{b}) + \lambda \sum_{d \in D} \|\mathbf{x}_d - \mathbf{x}'_d\|^2$$

where $\mathbf{x}'_d$ is the current altered image, can we turn this 4 into a 9?

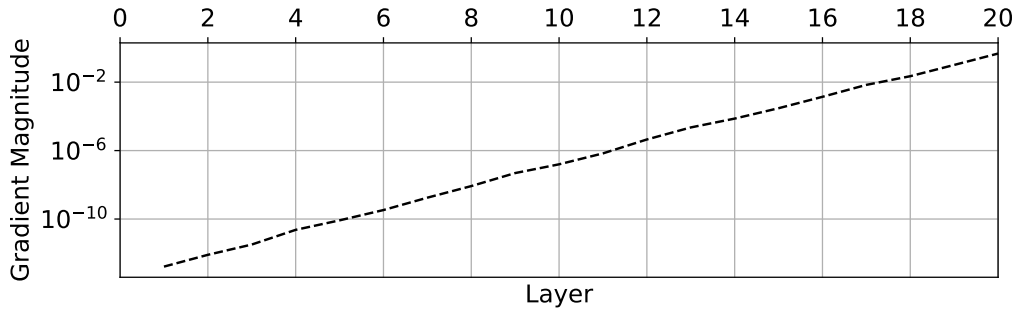


Vanishing Gradients

There are limits to the level of transformation that can be performed by a single layer of an MLP. As such, it is tempting to add more layers to the network to increase its expressive power. However, as we add more layers, we run into a problem: the gradients become very small, and thus the weights are not updated. This is called the *vanishing gradient problem.* Here are the derivatives of the most common activation functions from early neural nets: the hyperbolic tangent and sigmoid functions:

$$\begin{aligned} \tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \\ \frac{\partial \tanh(x)}{\partial x} &= 1 - \tanh^2(x) \\ s(x) &= \frac{1}{1 + e^{-x}} \\ \frac{\partial s(x)}{\partial x} &= s(x)(1 - s(x)) \end{aligned}$$

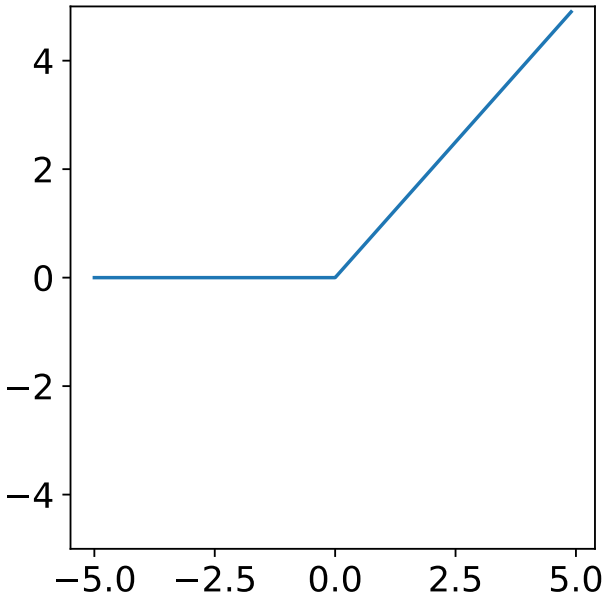
The derivative of both is always less than 1, and thus the gradient magnitude will decrease as we add more layers, as can be seen with the gradient for the sigmoid function:



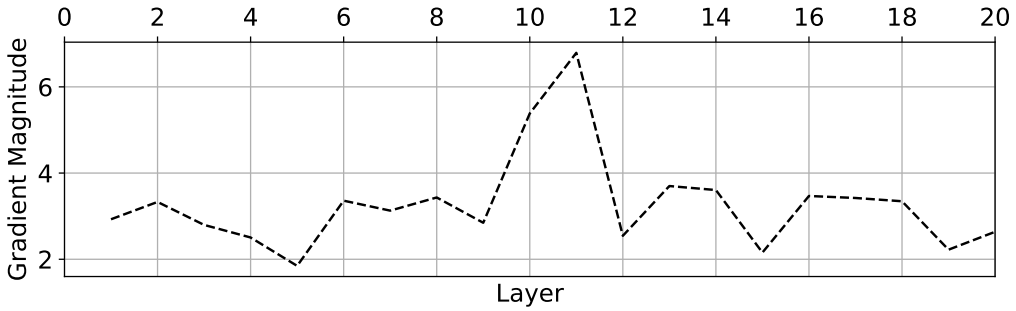
Rectified Linear Unit (ReLU)

One alternative to these activation functions is the *rectified linear unit* or ReLU:

$$f(x_i) = \begin{cases} x_i & x_i \geq 0 \\ 0 & x_i < 0 \end{cases}$$



Aside from the discontinuity at zero it is trivially differentiable and, importantly, does not saturate. This means that the gradient will not vanish as we add more layers.



Momentum

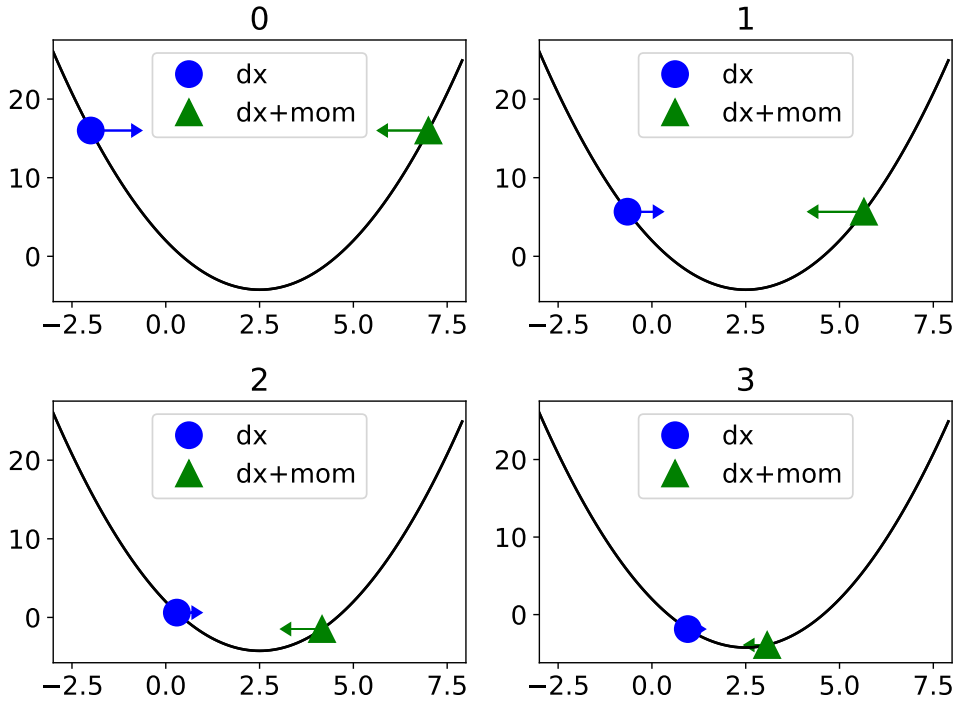
Before we move on to more complex deep learning structures, we will look at a few tricks of the trade. The first is a concept called *momentum*. A typical update in gradient descent is:

$$\mathbf{W}^{\tau} \leftarrow \mathbf{W}^{\tau-1} - \eta \frac{\partial L}{\partial \mathbf{W}^{\tau-1}}$$

where η is the learning rate and τ is the current training step. During learning it is often the case that there is a clear gradient direction, and we can move quickly along it. Conversely, if there is disagreement between subsequent gradients, we may want to move more slowly. We can achieve this by adding a momentum term:

$$\nabla \mathbf{W}^{\tau} = \frac{\partial L}{\partial \mathbf{W}^{\tau-1}} + \epsilon \nabla \mathbf{W}^{\tau-1}$$
$$\mathbf{W}^{\tau} \leftarrow \mathbf{W}^{\tau-1} - \eta \nabla \mathbf{W}^{\tau}$$

where ϵ is the momentum parameter.



Learning Rate Updates

As the network converges to a local minimum, it is often necessary to make smaller adjustments in the weight values, which translates to smaller step sizes during weight updates. A simple way of doing this is by using a learning rate that changes over time in a stepped fashion:

$$\eta^\tau = \eta^0 \gamma^{\lfloor \tau/\sigma \rfloor}$$

where the initial learning rate η^0 is multiplied by γ every σ steps.

Another method is to continuously alter the learning rate in inverse proportion to the number of steps:

$$\eta^\tau = \frac{\eta^0}{(1+\gamma\tau)^\rho}$$

where γ and ρ control the speed of learning rate decay.

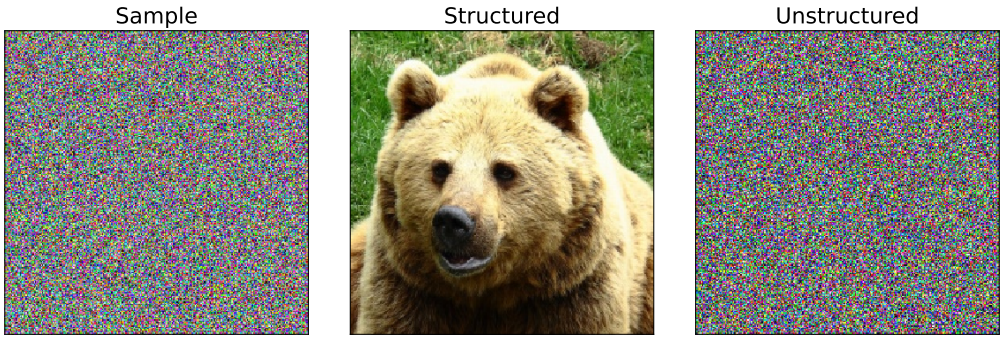
Another option is the popular Adam algorithm [5], which uses a running average of the gradient and its square to compute the learning rate:

$$\begin{aligned}
\mathbf{m}^\tau &= \beta_1 \mathbf{m}^{\tau-1} + (1-\beta_1)\frac{\partial L}{\partial \mathbf{W}^{\tau-1}} \\
\mathbf{v}^\tau &= \beta_2 \mathbf{v}^{\tau-1} + (1-\beta_2)\left(\frac{\partial L}{\partial \mathbf{W}^{\tau-1}}\right)^2 \\
\hat{\mathbf{m}}^\tau &= \frac{\mathbf{m}^\tau}{1-\beta_1^\tau} \qquad \hat{\mathbf{v}}^\tau = \frac{\mathbf{v}^\tau}{1-\beta_2^\tau} \\
\mathbf{W}^\tau &\leftarrow \mathbf{W}^{\tau-1} - \frac{\alpha \hat{\mathbf{m}}^\tau}{\sqrt{\hat{\mathbf{v}}^\tau} + \epsilon}
\end{aligned}$$

Reducing Model Capacity

At the moment our input transformations are performed by a fully connected layer, where each neuron in the output is connected to every input. While these layers are very expressive and can handle a wide range of potential inputs, they also require many parameters. In our example network from earlier trained on MNIST, the first layer (from the 784 pixels to the 64 dimensional latent space) required 50,240 parameters.

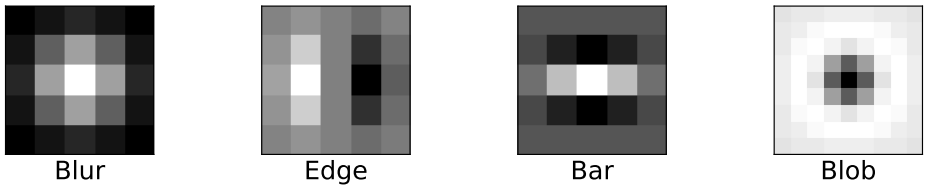
As a rule, the more parameters you have, the more capacity your network has to overfit to a dataset through memorisation. This hurts our overarching goal of generalisation. As such, we want to reduce the number of parameters in our network (if possible). One way of doing this is by taking advantage of known structure in the input data.



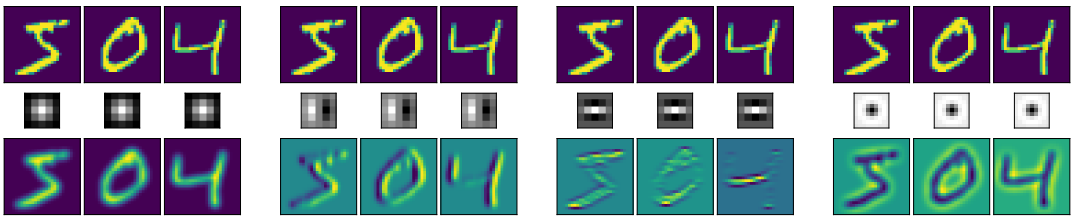
Above we see a random sample from the space of all images, a natural image, and the same image with its pixels randomly reordered. To an MLP, the image in the middle looks like the version on the right. Ideally, we would like to find a way to exploit the structure of natural images to reduce the number of model parameters.

Convolution

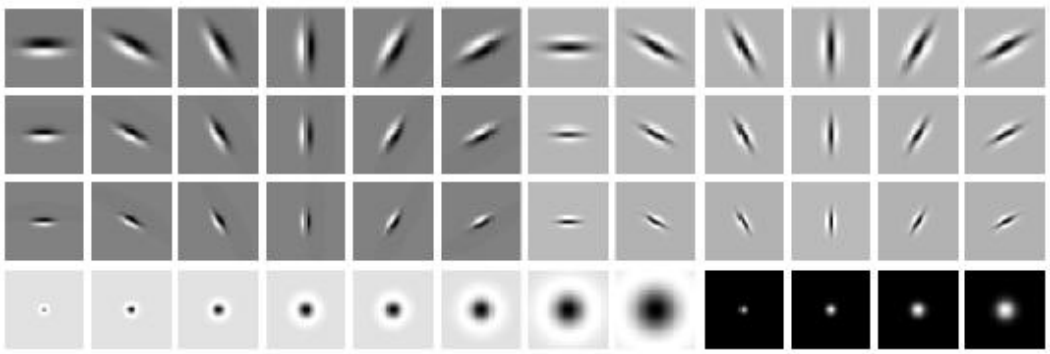
Let us briefly revisit the topic of convolution.



Filters are applied to an image by computing the dot product of a kernel patch and the image patch at each pixel location. This process is referred to as *convolution*, or more accurately cross-correlation.



There is evidence in biological vision for the existence of these filters in the visual cortex at different orientations and sizes, similar to this commonly used set of filters:



Learning Filter Kernels

Edge, bar, blur, and blob filters are examples of feature engineering. They transform an input image in much the same way as a neuron in an MLP, but requiring far fewer parameters. For example, a 5×5 filter has 25 parameters but can process an image of any size, whereas an MLP needs one weight per input pixel. However, these filters are handcrafted, and thus require human expertise. Ideally, we want to learn these filters from the data itself.

To enable this, we need to introduce an embedding function, $\mathcal{E}$, which extracts image patches in row column order and embeds them as columns in a matrix:

$$\mathbf{x} = \begin{pmatrix} a & b & c & d & e \\ f & g & h & i & j \\ k & l & m & n & o \\ p & q & r & s & t \\ u & v & w & x & y \end{pmatrix} \quad \mathbf{X} = \begin{bmatrix} a & b & \dots & g & \dots & m \\ b & c & \dots & h & \dots & n \\ c & d & \dots & i & \dots & o \\ f & g & \dots & l & \dots & r \\ g & h & \dots & m & \dots & s \\ h & i & \dots & n & \dots & t \\ k & l & \dots & q & \dots & w \\ l & m & \dots & r & \dots & x \\ m & n & \dots & s & \dots & y \end{bmatrix}$$

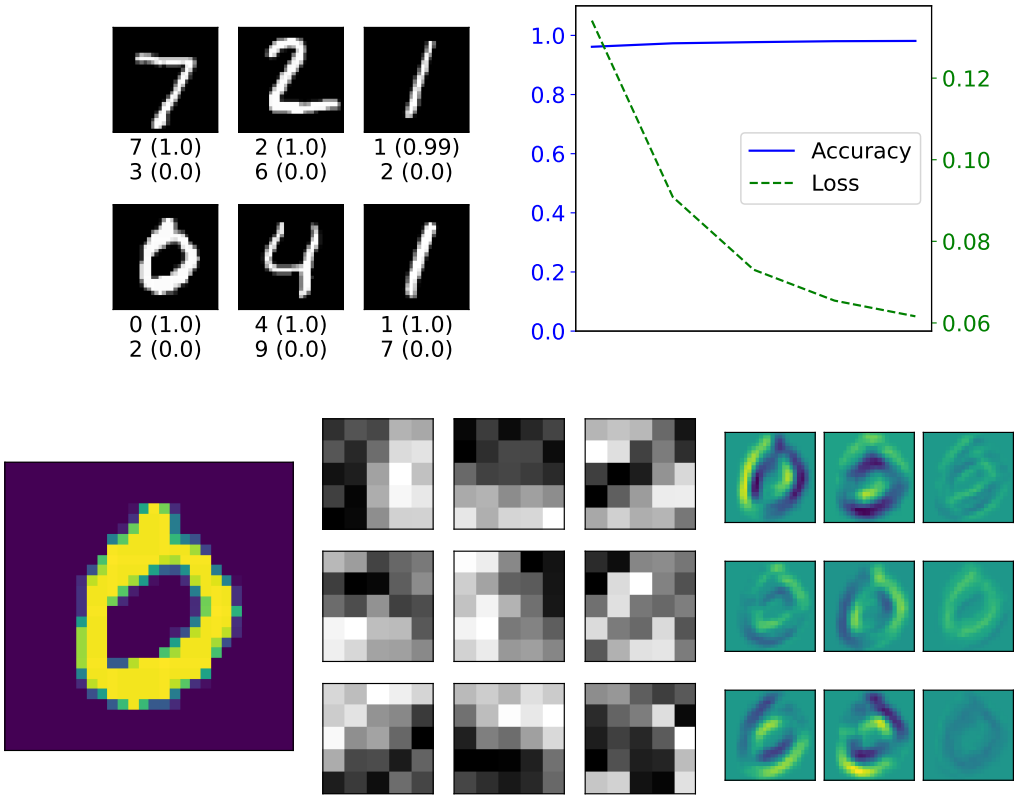
We can then store the filter kernel weights in a matrix and use matrix multiplication to perform the convolution:

$$\mathbf{W}_i = \begin{bmatrix} w_{000} & w_{001} & w_{002} & w_{010} & w_{011} & \dots & w_{021} & w_{022} \\ w_{100} & w_{101} & w_{102} & w_{110} & w_{111} & \dots & w_{121} & w_{122} \end{bmatrix}$$

$$C(x) = \mathbf{WX}$$

CNN

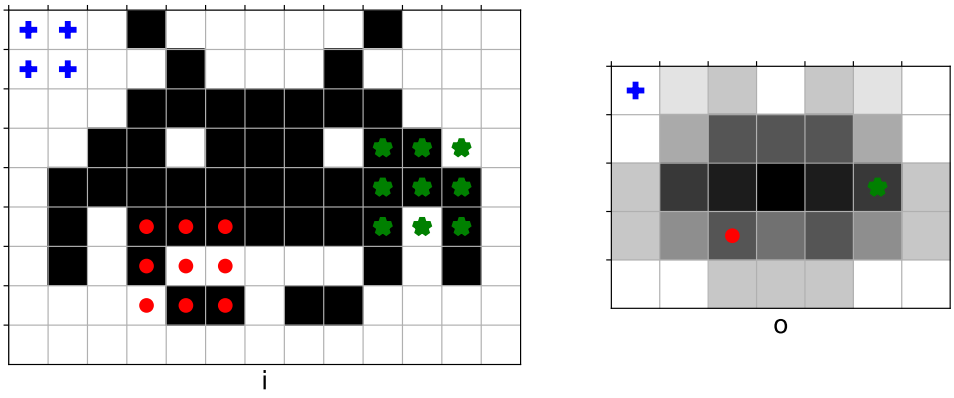
Convolution Neural Nets (CNNs) use trainable filter banks in exactly this way as a replacement for the initial fully connected layers in an MLP. We will begin by taking the MLP we trained on MNIST and replacing its initial layer with a convolutional layer with 9 filters.



These learned filters are recognisable to us as oriented bars, edges, and corners, but have been tailored to the dataset and object. A different objective function or dataset may result in different features. However, we have created a new problem. In the original MLP the final layer had $64 \times 10 = 640$ weights. Since each convolution produces an image, the final layer here has $9 \times 24 \times 24 \times 10 = 51,840$ weights. We need a way to reduce the dimensionality of the filter layer output.

Subsampling

One way we can reduce the dimensionality of our convolutional layer is by *subsampling* the image in space. Layers that perform this subsampling are called *pooling* layers. One approach is average pooling, where we use the average value in a window:



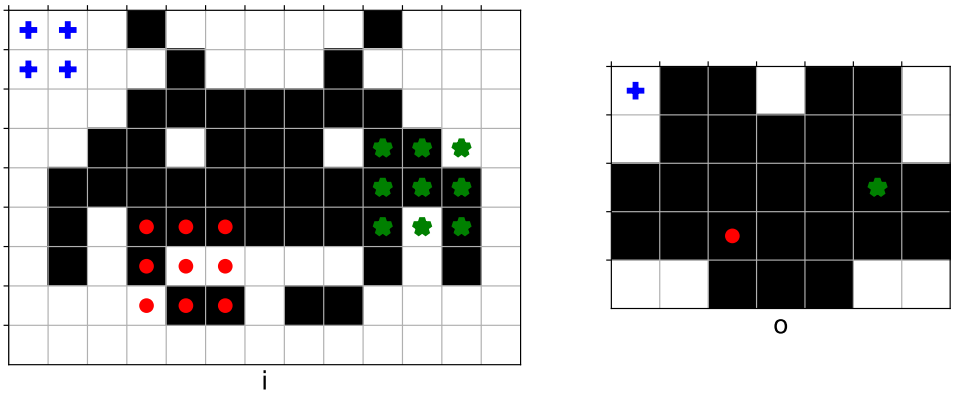
As can be seen above a filter of size 3 is applied at a *stride* of 2, though the size and stride can be altered in practice as needed by the data. This pooling acts as a *non-parameterised* layer in the neural network, much like non-linearities. An output value is computed as follows:

$$o[r, c] = \frac{1}{9} \sum_{k=0}^{2} \sum_{l=0}^{2} i[2r + k - 1, 2c + l - 1]$$

This is equivalent to using the embedding function $\mathcal{E}$ and then multiply by a uniform weight matrix where every value is equal to $\frac{1}{9}$.

Subsampling, cont.

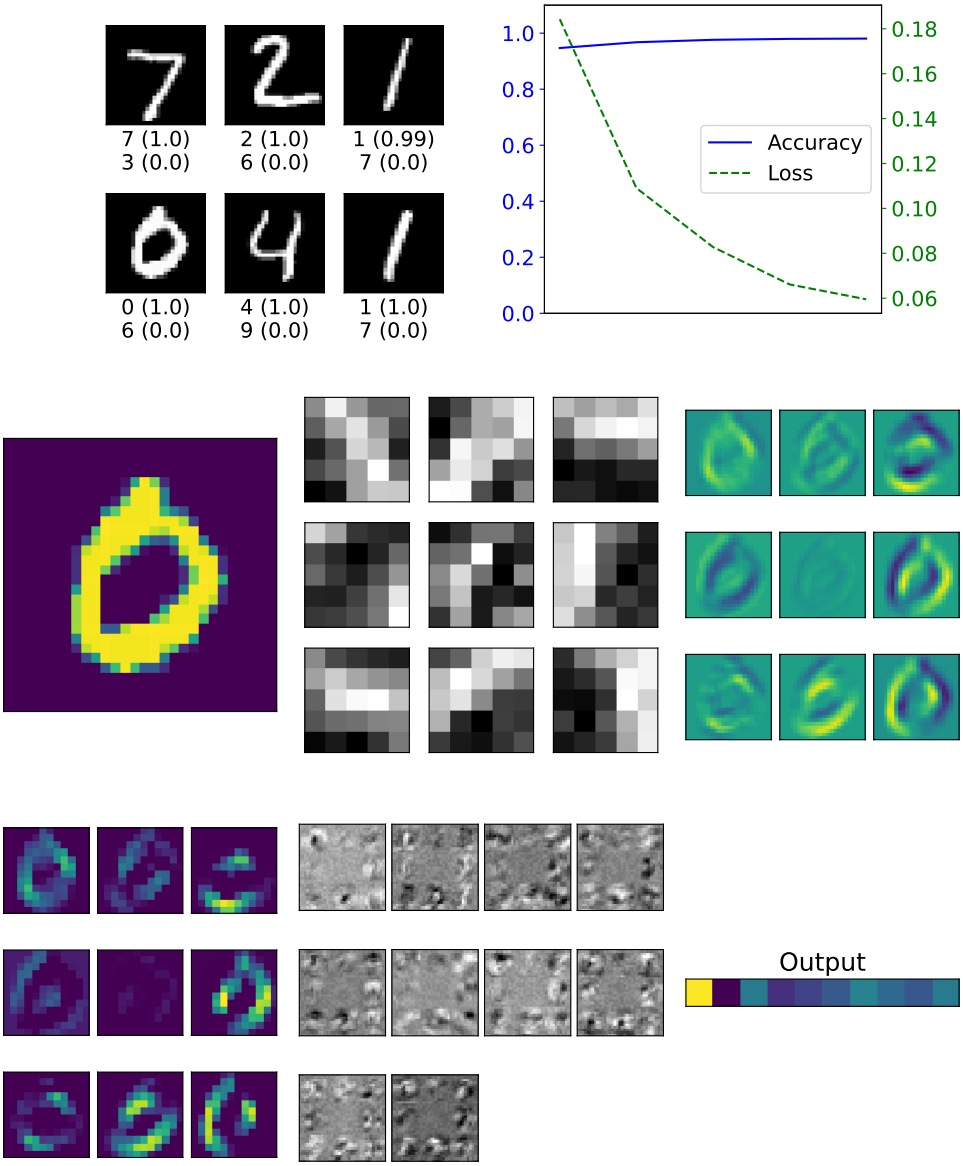
An alternative to average pooling is maximum, or max, pooling. As with average pooling, the max pooling operator has a set size and is applied at a predetermined stride. However, instead of taking the average value of all of the pixel values it selects only the maximum as its output value:



This creates a slight difficulty during backpropagation, as the partial derivatives passing backwards through the operator only apply to a single input pixel. As such, we need to keep track of which pixel was the maximum in the forward pass and only apply the gradient to that pixel in the backward pass.

CNN with Pooling

When we incorporate max pooling with a window size of 2 into our CNN we reduce the dimensionality of the filter layer output by a factor of 4.



The max pooling reduces the output image size of the filter layer from 24×24 to 12×12 but preserves the spatial feature information in the image. We are outgrowing MNIST, however, and need to move on to a more complex dataset.

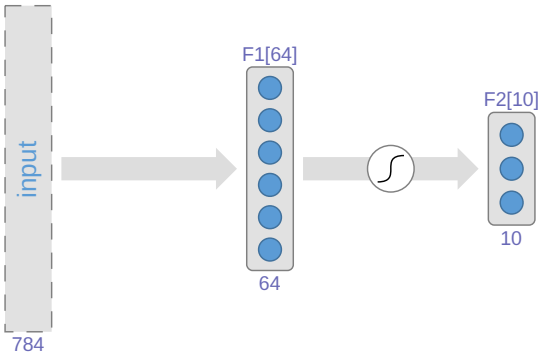
CIFAR-10

CIFAR-10, while it has many of the same characteristics as MNIST, is a considerable step up in difficulty. It is drawn from the "80 million tiny images" dataset which consists of (almost) 80 million 32×32 RGB images downloaded from the internet and labeled with one of 75,062 non-abstract nouns in English.

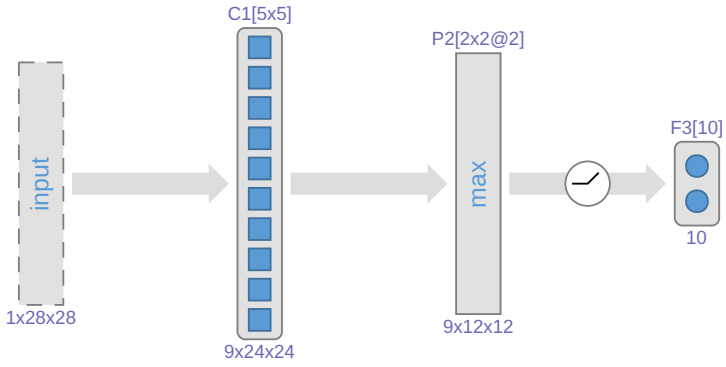
Named after the Canadian Institute for Advanced Research, it consists of 60,000 images gathered from this larger set by Alex Krizhevsky, Vinod Nair and Geoffrey Hinton. These images belong to 10 classes: (in order from left to right above) aeroplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. As can be seen, the dataset contains a lot of intra-class variance, in addition to complications brought about by colour and the much larger variety inherent in non-curated natural images.

DNN Architecture

Before we begin to create more complex DNNs architectures we need a way to diagram and visualize them. Here is a diagram of our first MLP:



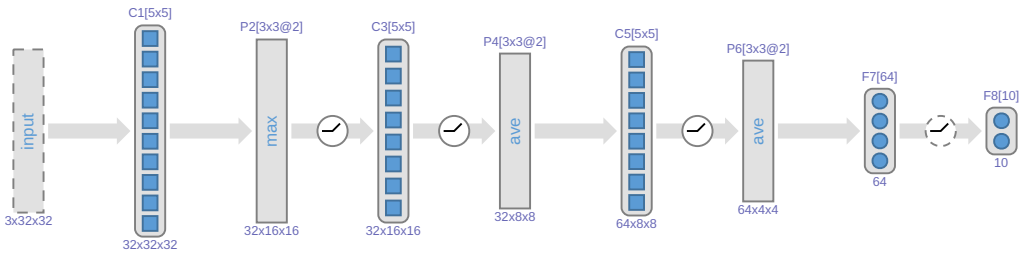
Note how we denote the output size in each layer at the bottom, and use a circle to indicate that they are perceptrons, *i.e.* fully connected, with the number of neurons in brackets. Also note how we depict the activation function. Here is the CNN we just trained:



Here, note the use of squares to indicate that it is a convolutional neuron. Also note the named pooling layers, and the patch notation above the convolutional layers. We will use this notation moving forward as we discuss more complex DNN architectures.

Anatomy of a CNN

This is a diagram of a CNN we will train on the CIFAR-10 dataset:



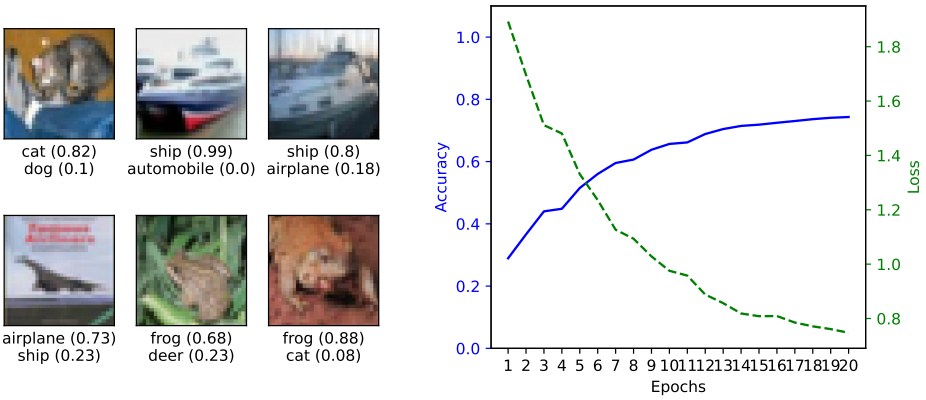
C1 Extracts low-level features in the image.
P2 Provides some flexibility of location
C3 Looks for parts that are combinations of features
P4 Smooths the part responses before subsampling
C5 Finds structures that are built from parts
P6 Smooths and subsamples the structural responses.
F7 Sub-category latent space
F8 Final classifier

The dashed border around the final activation indicates that it incorporates *dropout*. Dropout is a technique for regularising the network by randomly dropping neurons during training. It applies the following transform:

$$\mathbf{y} = \mathbf{x} \odot \mathbf{m}$$
$$\mathbf{m} \sim \text{Bernoulli}(p)$$

where $\mathbf{m}$ is a binary mask with probability p of being 1. This forces the network to learn redundant representations, which in turn makes it more robust to noise and overfitting. Dropout is only used during training, and is turned off during testing ($p = 1$).

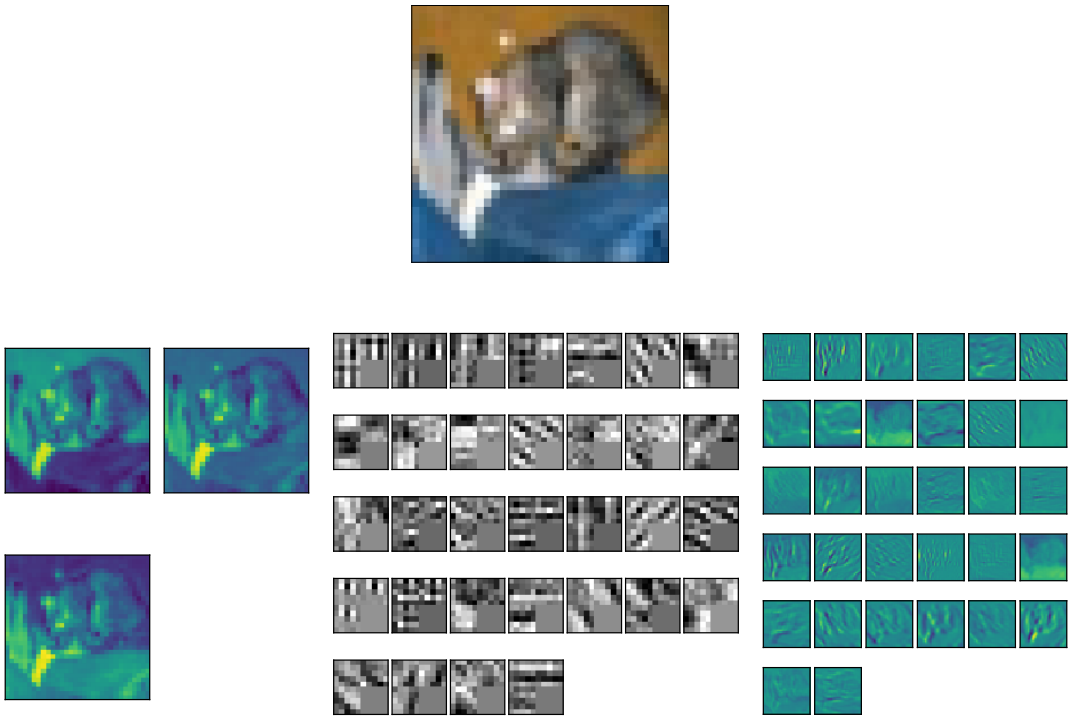
CIFAR-10: DNN



The DNN does a fairly good job of classifying these images, though CIFAR-10 and its big brother CIFAR-100 are not easy. Visualising what this network is learning is difficult, but we will pick it apart layer by layer.

CIFAR-10: C1

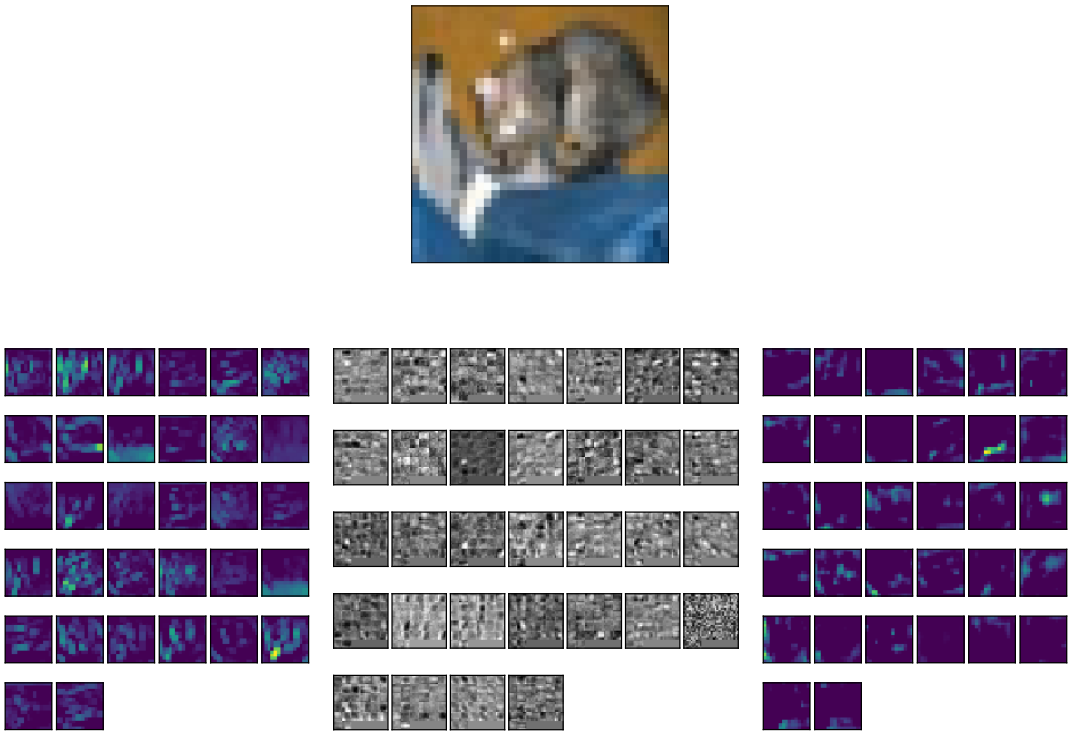
The first layer learns low-level image features such as those you have seen before, but across three feature channels (as these are colour images):



Note how some filters are pulling out a particular colour, while others are looking at structure.

CIFAR-10: P2 → C3

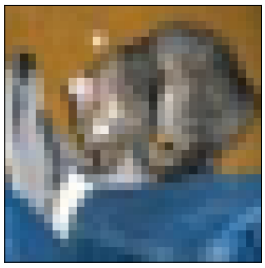
C3 receives the output of P2 as input. Note how the max pooling creates pinpoints of high activation. At this point visualising the filters themselves becomes less helpful, as they are defined over all 32 feature channels. However, we can still look at the filter responses:

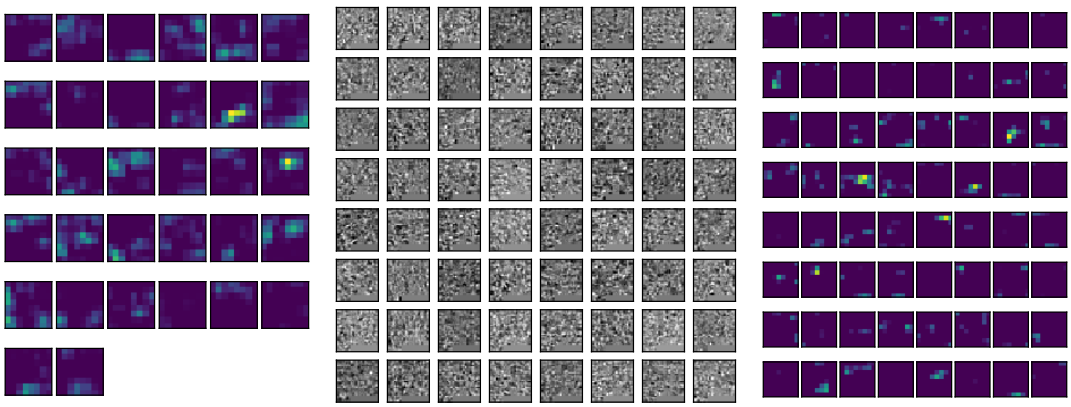


We can definitely see here similar localized pinpoints of activation, but instead of localizing texture or colour we are locating combinations of low-level features.

CIFAR-10: P4 → C5

As we move along, images get smaller but each pixel stores an increasing amount of semantic information. P4 smooths and subsamples the part responses, and then C5 is looking for structures that are built from those parts.





Deep CNN Training

The following is a typical procedure for training a deep CNN on a dataset:

1. Divide the dataset into training D, validation V, and test T partitions.
2. Compute the "mean" image of D, and subtract it from all images.
3. Scale all images so that pixel values are in the range $[-1, 1]$.
4. Design a network architecture, or adapt a known good architecture to the dataset.
5. Select an optimisation algorithm (*i.e.* a version of SGD)
6. For a pre-determined set of epochs, do the following:
   (a) Randomly sample (without replacement) B images from D.
   (b) Compute $\nabla \mathbf{W}^\tau$ for all $\mathbf{W}$ in the model
   (c) Update all $\mathbf{W}^\tau \rightarrow \mathbf{W}^{\tau+1}$
   (d) Once all images in $\mathbf{D}$ have been seen (*i.e.* when an epoch is complete), evaluate the network on $\mathbf{D}$ and $\mathbf{V}$ to monitor changes in the loss
7. Evaluate final performance on $\mathbf{V}$, and repeat the above with different optimisation hyperparameters as necessary
8. Retrain the network using $\mathbf{D} \cup \mathbf{V}$ with the final hyper-parameters, and evaluate on $\mathbf{T}$

Data Augmentation

When it comes to deep networks, the only thing better than a lot of data is even more data. However, gathering supervised data is very expensive. One way to get more data on the cheap, however, is via *data augmentation*, whereby on each epoch each training image is turned into several additional images by way of translation, cropping, and horizontal reflection:

In this way, one training image can become ten. Another common usage of data augmentation is during the testing phase, whereby instead of simply computing $P(c|\mathbf{x})$ from a single image, $\mathbf{x}$ is altered as shown above and then each version of $\mathbf{x}$ is presented to the network. In this scenario, the predicted class is then:

$$c = \underset{i}{\operatorname{argmax}} \left| \left\{ \mathbf{x}_a \mid i = \underset{j}{\operatorname{argmax}} P(j \mid \mathbf{x}_a) \right\} \right|$$

This can result in a significant improvement in performance regardless of the network architecture.

Batch Normalisation

Batch normalisation [4] is a technique that is essential to training the deepest neural net architectures, but also useful in general for stabilising and accelerating the training process in all circumstances. The concept is straightforward: if we knew the statistics of each layer's output over the entire dataset, we could normalise those outputs so that all output vectors had zero mean and unit variance. In mathematical terms, we want to do the following:

$$\mu_D^i = \frac{1}{|D|}\sum_{d\in D}\mathbf{y}_d^i$$
$$\sigma_D^i = \sqrt{\frac{1}{|D|}\sum_{d\in D}\left(\mathbf{y}_d^i - \mu_D^i\right)^2}$$
$$\hat{\mathbf{y}}_d^i = \frac{\mathbf{y}_d^i - \mu_D^i}{\sigma_D^i + \epsilon}$$

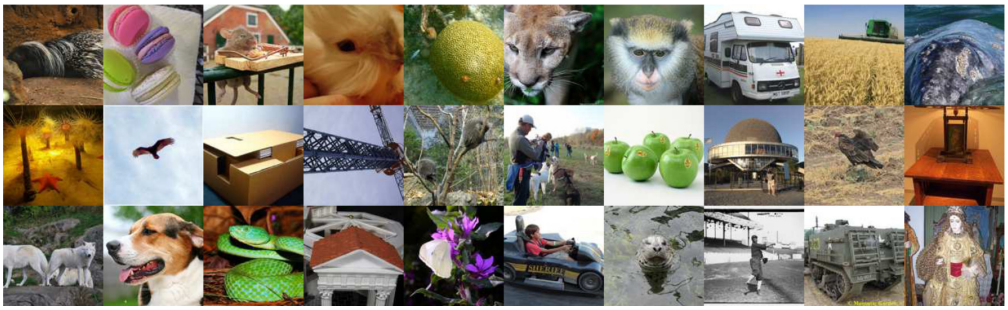
where i indexes the layers of the DNN.

The problem is that the values of μ_D^i and σ_D^i change during training as the network changes. We overcome this by using batching to estimate these values *i.e.* by computing them per batch during training. Once the network is trained, we can then compute the true values and use those during testing. We can even learn training parameters for scaling and shifting the normalised vectors:

$$\mathbf{z}^i = \gamma^i\hat{\mathbf{y}}^i + \beta^i$$

ImageNet

Now we will look at the first truly large-scale classification dataset: ImageNet. It consists of over 14 million images mapped to around 20,000 labels. The images are extremely varied, coming from a variety of difference sources all over the internet, and have each been hand-labelled by one or more human labellers.

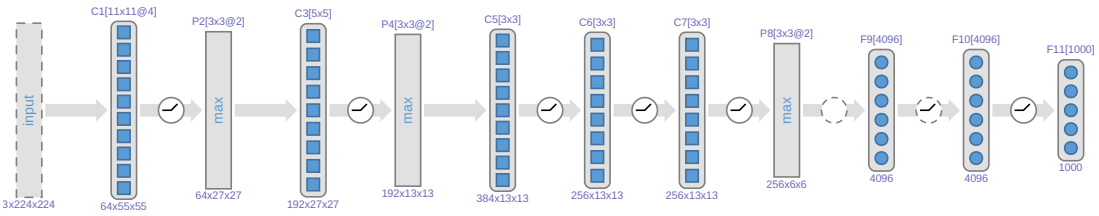


For classification tasks, the ILSVRC 2012 subset is the most commonly used. It consists of 1.2 million images, with 1,000 object classes. The dataset is split into training, validation, and test sets, with 1000, 50, and 100 training, validation, and test images per class, respectively. Performance is measured using top-1 and top-5 error rates, where the top-5 error rate is the percentage of test images where the correct label is not in the top 5 predicted labels (lower is better).

Here are some samples from the "Handheld computer", "Dandie Dinmont Terrier", and "Teapot" classes. Note the extreme variability in viewpoint, lighting, and background.

AlexNet

When ImageNet was first introduced, it was prohibitively expensive to train a neural network on it. However, in 2012 Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton introduced AlexNet [7], a CNN architecture that was able to achieve a top-5 error rate of 15.3% on the ILSVRC 2012 dataset, a 10.8% improvement over the previous state-of-the-art. To achieve this they harnessed a new technology for use in training DNNs.



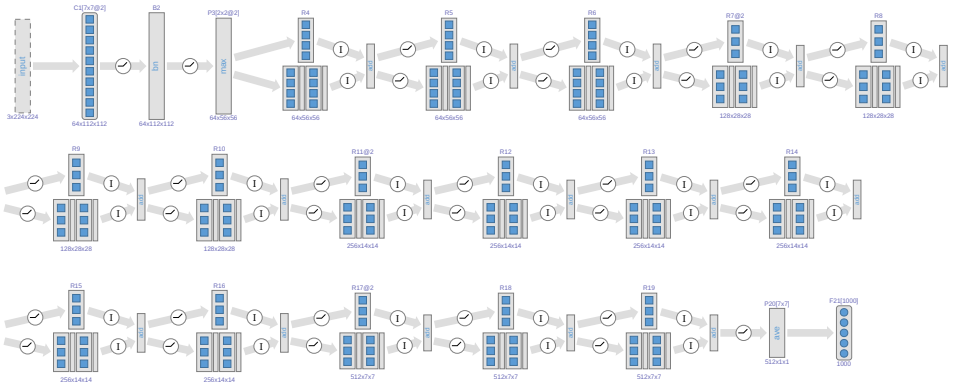
Graphical Processing Units, or GPUs, were originally designed for graphically intensive workflows, like video games. In particular, they are specialised towards the execution of per-pixel code in which linear algebra is used to project 3D model information to the camera plane. One can coerce the hardware into performing other computationally intensive image tasks, *e.g.* those in computer vision. As a demonstrator, see this implementation of Canny edge detection as using GPU shaders:

`https://matajoh.github.io/canny/`

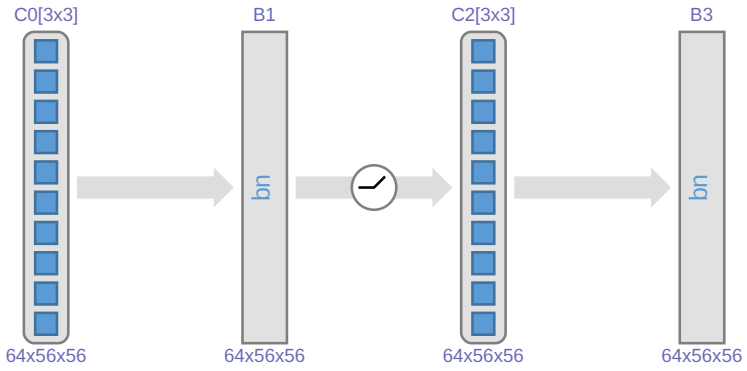
Krizhevsky *et al.* realised that this same hardware could be used to perform the matrix multiplications required by CNNs, and that the parallel nature of GPUs would allow them to train much faster than on CPUs.

Residual Networks

Even with all the tricks we have introduced so far, very deep networks fail to train due to a version of the vanishing gradient problem. In 2015, Kaiming He *et al.* introduced Residual Networks [2], a new architecture that was able to achieve a top-5 error rate of 3.57% on the ILSVRC 2012 dataset despite being 152 layers deep. Here is the 34-layer configuration:



The key contribution is the repeating *residual block* within this diagram. It has two branches: an identity branch which copies the input, and the following miniature network:



The identity branch acts as a shortcut, allowing the network to ignore large blocks of its capacity if they are not needed. In essence, the network grows deeper each time a residual block saturates during training.

ImageNet: ResNet

Here are some top 5 predictions on the ILSVRC 2012 validation set for a trained ResNet model with a depth of 50:

1. tractor
2. harvester
3. thresher
4. snowplow
5. lawn mower

1. tiger
2. tiger cat
3. jaguar
4. consomme
5. platypus

1. barn
2. alp
3. birdhouse
4. stone wall
5. black grouse

1. web site
2. castle
3. analog clock
4. book jacket
5. comic book

1. stage
2. dock
3. toyshop
4. balance beam
5. lifeboat

1. macaw
2. sulphur-crested cockatoo
3. African grey
4. lorikeet
5. screen

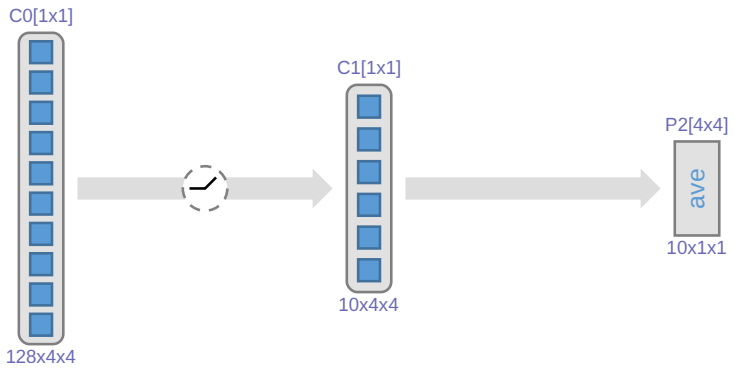
1. punching bag
2. oxygen mask
3. cash machine
4. turnstile
5. stretcher

1. laptop
2. notebook
3. computer keyboard
4. desktop computer
5. space bar

Reducing Parameters, redux

One feature the CNNs we have examined so far share is one or more fully connected layers at the end which perform the desired task. In this paradigm, the convolutional layers are purely for latent space embedding. As such, the final convolution layer gets flattened into a 1D vector, resulting in a phenomenon where the penultimate layer of a CNN often has more trainable parameters than the rest of the network combined.

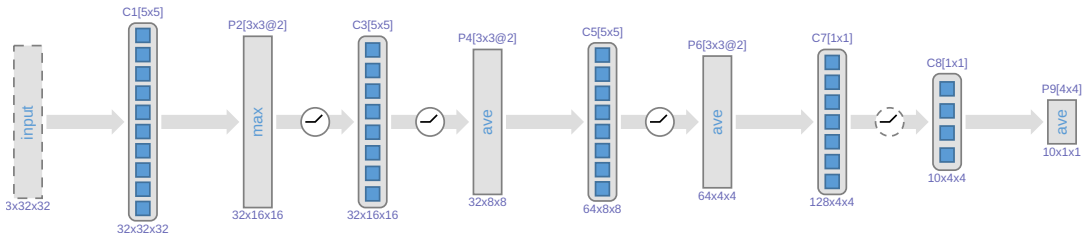
Recall that we want, wherever possible, to reduce the number of parameters in order to increase generalisation. One way we can do this is to embed a miniature fully connected layer into the convolutional layers themselves using a technique called Network in Network [8] or 1×1 convolution. Here is a sample NiN unit which can be used to replace the final fully connected layers of the DNN we trained previously:



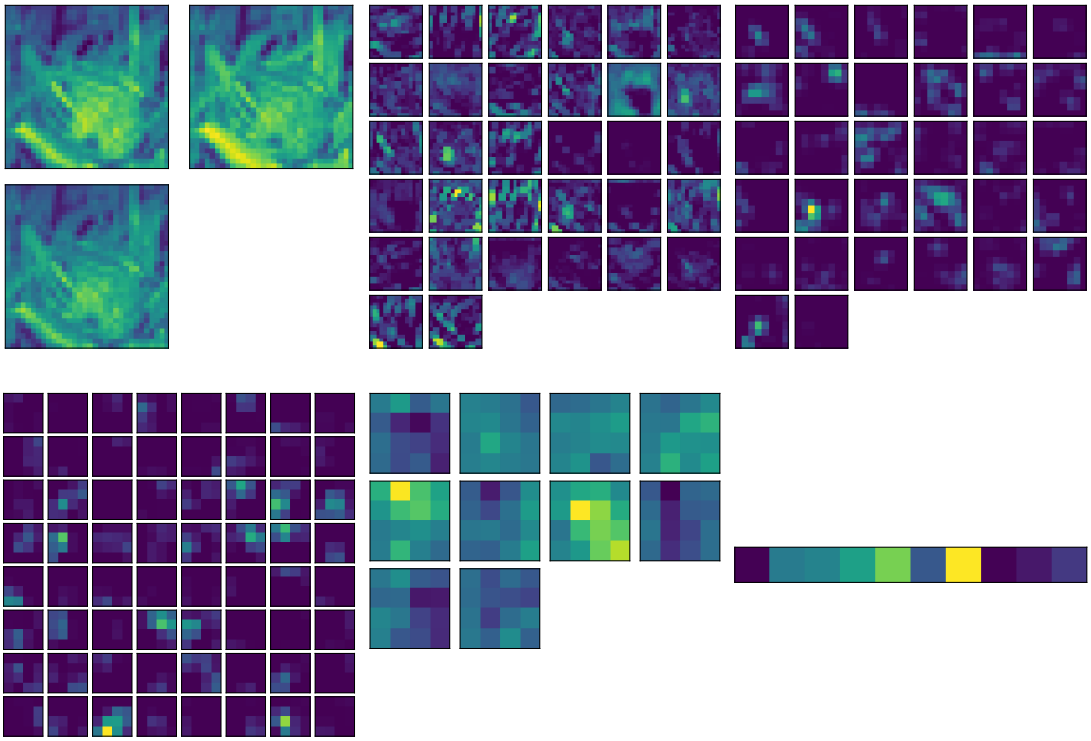
Since the kernel is applied to a single pixel at a time, two or more of these layers in sequence act as if a small MLP was applied across the filter channels. By pooling the result we can build a system which predicts classes for the image without any fully connected layers at all, which is called a Fully Convolutional Network [10].

Semantic Segmentation

We can modify our original DNN to use Network in Network instead of fully connected layers:



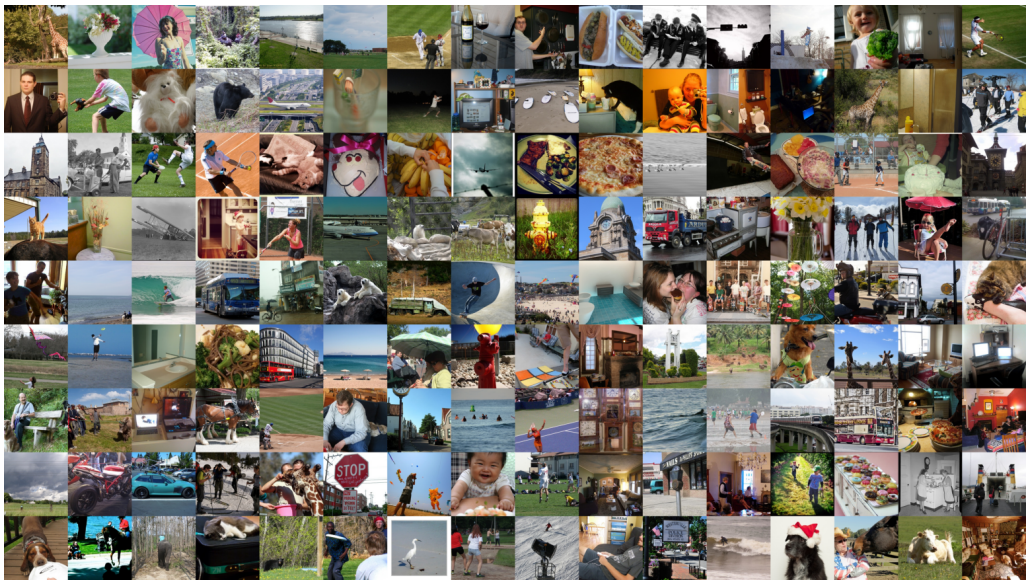
For a particular input image, we can visualise the outputs of the layers like so:



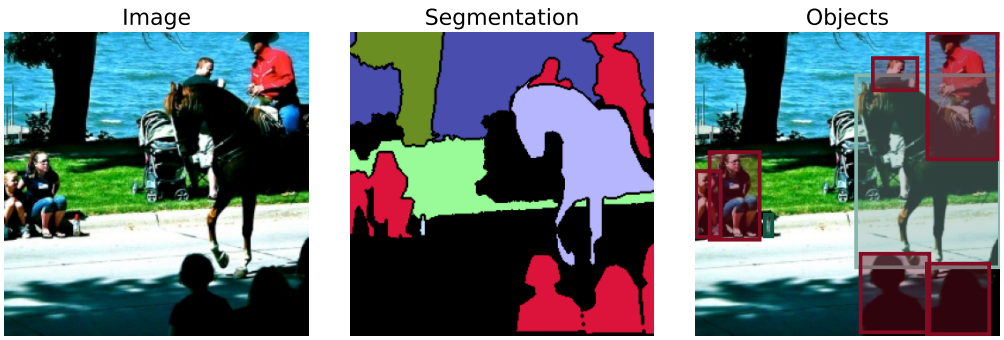
While the network uses a global average pooling layer to make a single prediction, if we replace this pooling layer with an interpolation layer we can get a prediction for each pixel in the image. This new task is called *semantic segmentation*.

COCO

As is often the case, a new task means a new dataset. An important benchmark here is the Common Objects in Context Dataset, or COCO [9].

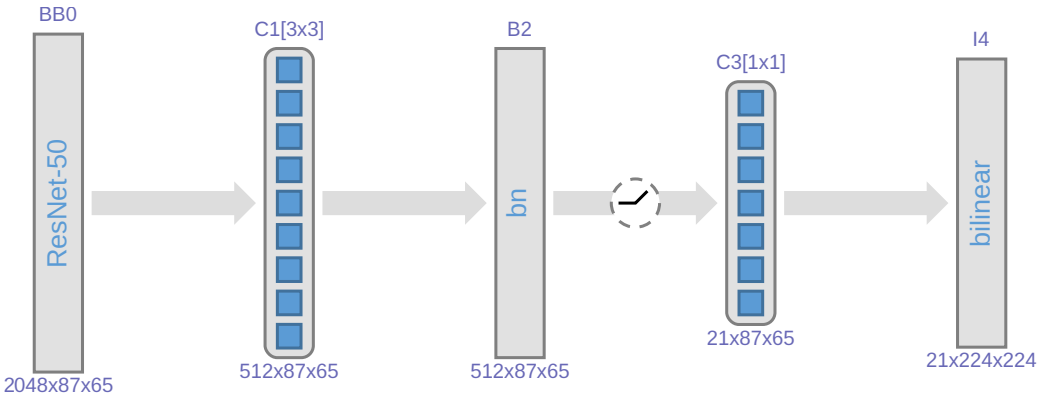


It consists of 330K images which contain 1.5 million object instances. They depict scenes of stuff drawn from 91 classes co-occurring in typical ways. In particular, the stuff in the images has been hand segmented such that there are pixel-level ground truth labels that can be used during training. Here is a sample image with its segmentation and objects:

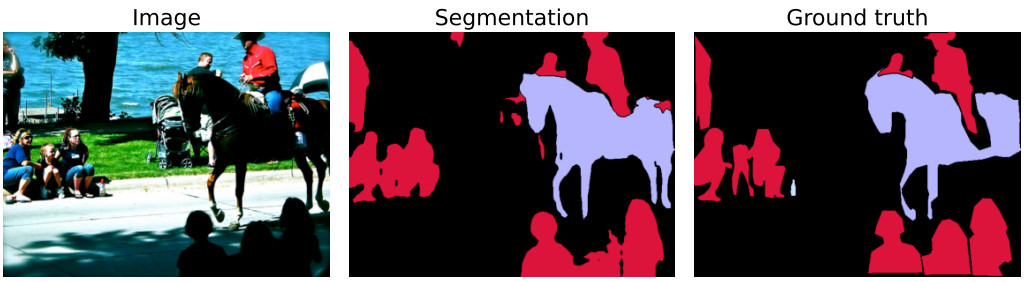


FCN

One way to use an FCN to perform semantic segmentation is to replace the final convolutional layers of a typical classifier with convolutional layers, and then scale the result back up to the original resolution using bilinear interpolation. Here is a sample FCN architecture:

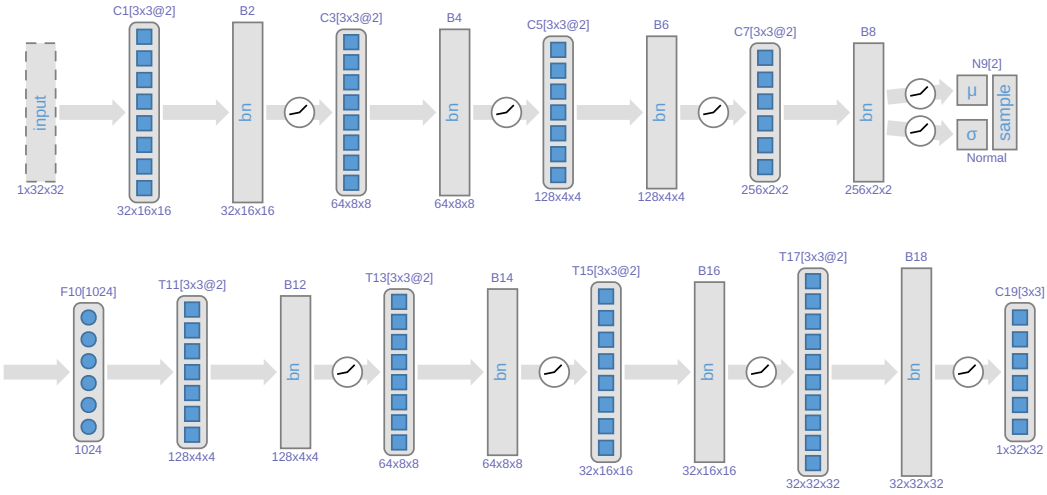


Note that we are using ResNET-50 as a *backbone* network, meaning that its weights are pretrained (in this case on ImageNet) and either fine-tuned or left alone during training. Additionally, we have removed striding from a few convolutions in the residual blocks so that we do not lose too much resolution. The result is a network that can produce a segmentation mask for an image:



Autoencoding

Autoencoding is a method of encoding and decoding images to a latent space. During training, we use the image as both input to an encoder network and as target for a decoder network, with a latent space *bottleneck* in between.



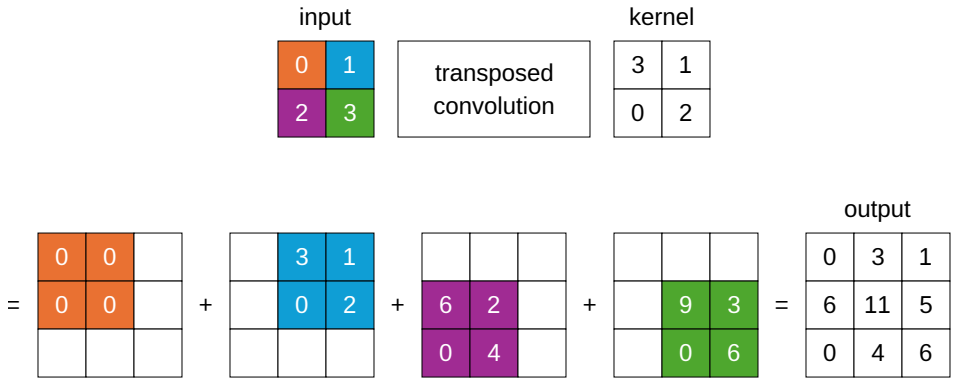
The architecture above depicts a *variational autoencoder*[6]. The final layer of the encoder outputs two vectors, μ and σ. The system samples from $\mathcal{N}(\mu, \sigma)$ and uses the result as the latent space input to the decoder. The decoder uses transposed convolution to scale up the image (denoted as T). This requires a new loss: Evidence Lower Bound, or ELBO:

$$\mathcal{L} = \min \left(\mathbb{E}_{\mathbf{z} \sim q(\mathbf{z}|\mathbf{x})} \left[\log p(\mathbf{x}|\mathbf{z}) \right] - D_{KL} \left(q(\mathbf{z}|\mathbf{x}) || p(\mathbf{z}) \right) \right)$$

The first term is a *reconstruction loss*, which will train the generative distribution $p(\mathbf{x}|\mathbf{z})$ (*i.e.* our decoder) to transform the latent vector $\mathbf{z}$ into our image $\mathbf{x}$. The second term is a *regularisation loss*, which trains the $q(\mathbf{z}|\mathbf{x})$ distribution (*i.e.* our encoder) to generate $\mathbf{z}$ that resemble samples from $p(\mathbf{z})$, which in this case is normal.

Transposed Convolution

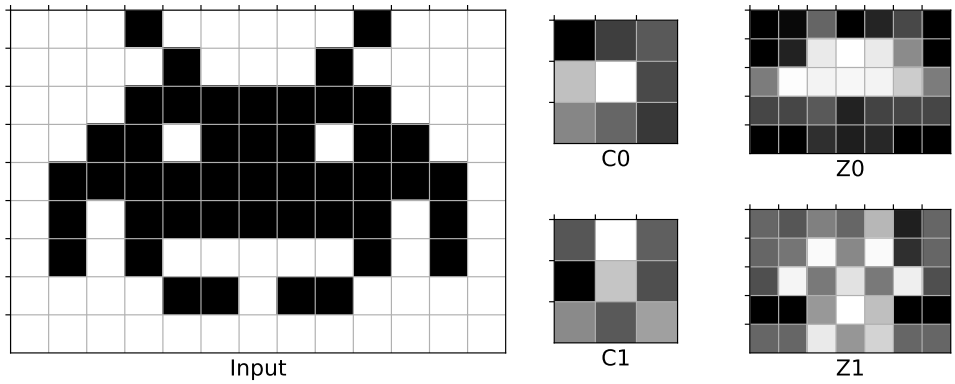
In CNNs, each convolutional layer with a stride greater than 1 trades *spatial* information (*i.e.* resolution) for *semantic* information (*i.e.* channels). In an autoencoder, we want to reverse this process, that is to trade semantic information for spatial information so that we can arrive back at the resolution and pixel depth of the original image. Layers that do this are said to perform *transposed convolution*. Whereas a convolution takes the dot product of a filter and an image patch to produce a single value, a transposed convolution scales a filter by a value and then adds it to a patch of the output image.



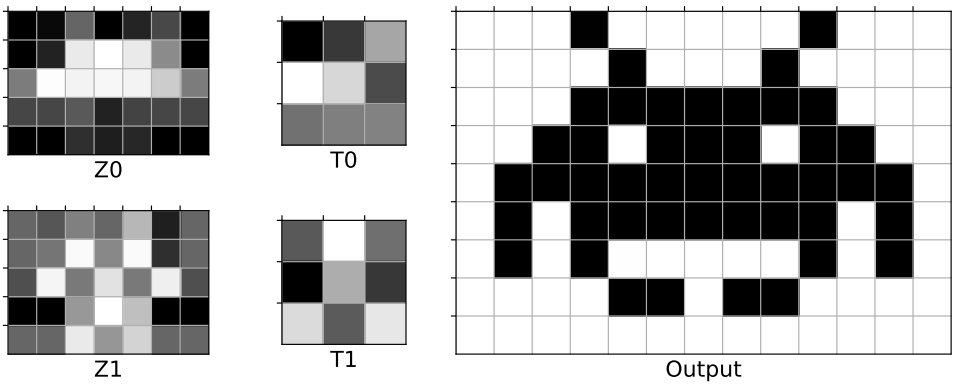
Here we see a 2×2 filter being applied with a stride of 1 to a $2x2$ image. The result is a 3×3 image. The colors correspond to the image pixel which is being used to scale the filter. The resulting images are added together to create the final upscaled image.

Transposed Convolution, cont.

Now we will take a look at this in practice. First, we have a convolutional layer with 2 3×3 filter kernels being applied to an image with padding of 1 and stride of 2.

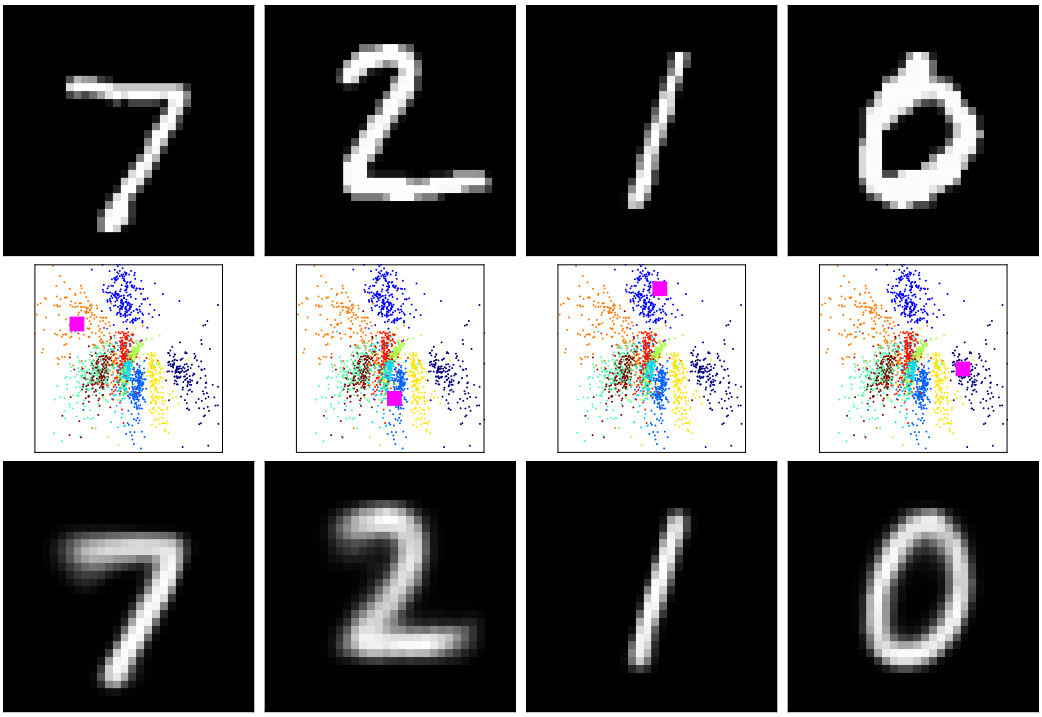


Note how each channel of the z contains different information. We can then scale back up to the original image using a transposed convolution with a single $2 \times 3 \times 3$ kernel:

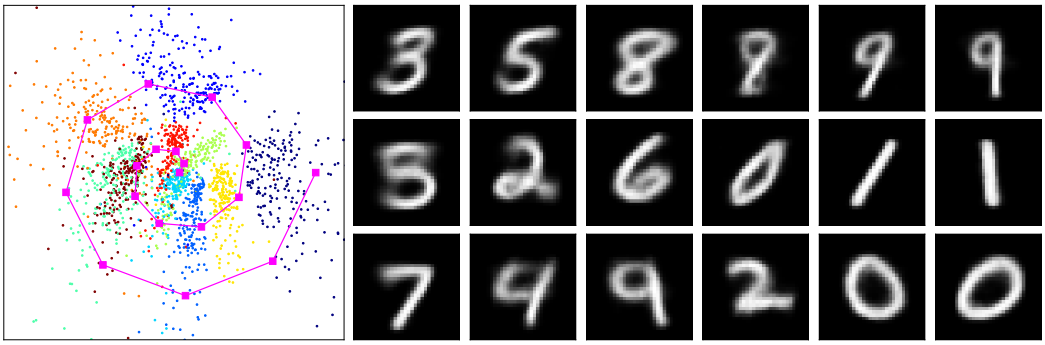


Sampling the Latent Space

This is the first task we have considered that does not require supervision, *i.e.* human-provided labels. Instead, we can train this system in an *unsupervised* manner, since the image is both the input and output of the network. Here is an example of some images, their latent points, and reconstructions:

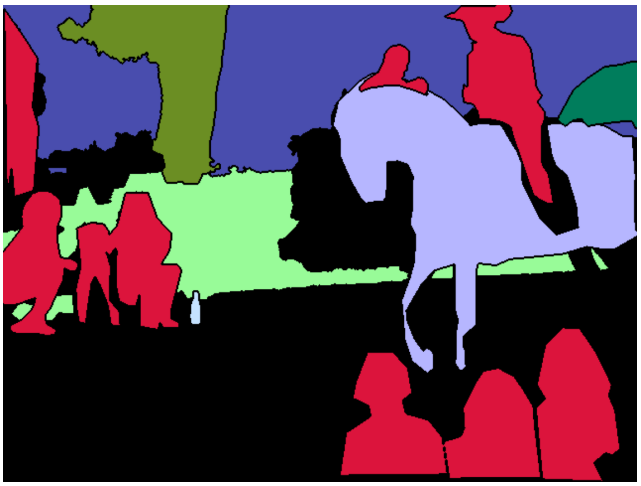


Because we have constrained samples from $q(\mathbf{z}|\mathbf{x})$ to resemble $p(\mathbf{z})$, we can sample from p instead and generate never-before-seen images:

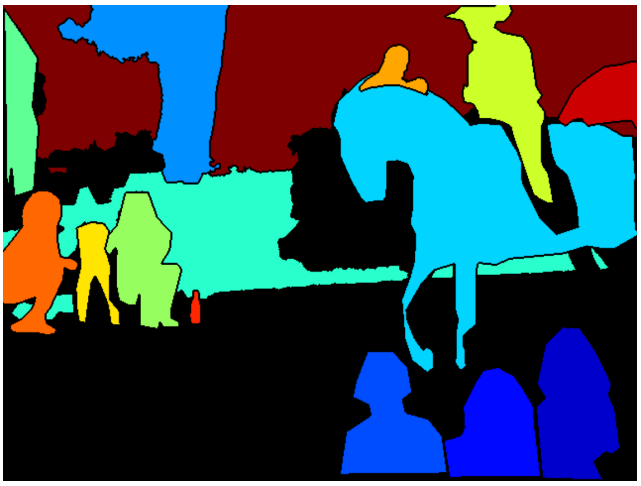


Object Detection

We saw with semantic segmentation how a CNN could label each pixel with a distribution over classes. One drawback of this approach is that if there are multiple members of a class, *e.g.* people, they will all be given the same label:



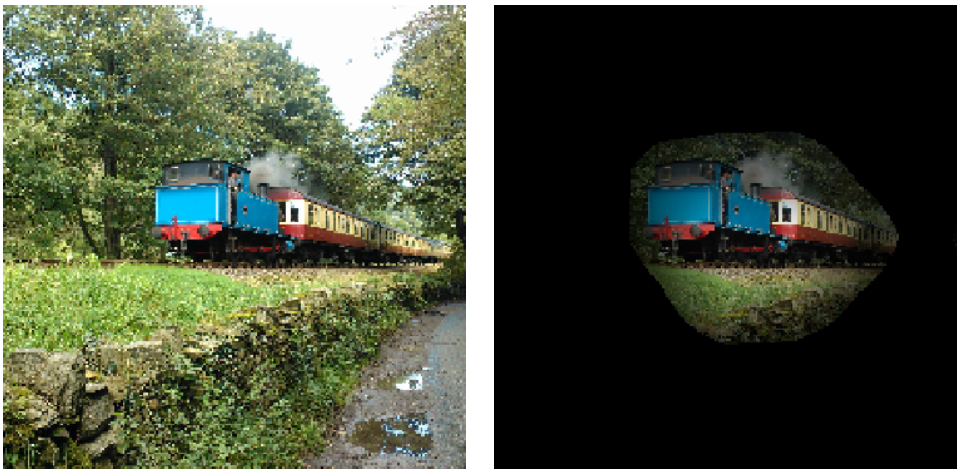
If we want to give them different labels, this is called *instance segmentation* or more generally, *object detection*:



The COCO dataset, in addition to having per-pixel labels, also contains bounding boxes around each object instance. This allows us to train a CNN to perform object detection.

Salience

One way to detect objects in an image is to train a classification network and then find which pixels contribute to each output class. This is called *salience* or *attribution*. While this is not a perfect method, it is fast and can be used to explain the network's predictions. For example, here is an image of a steam locomotive and its salience mask:

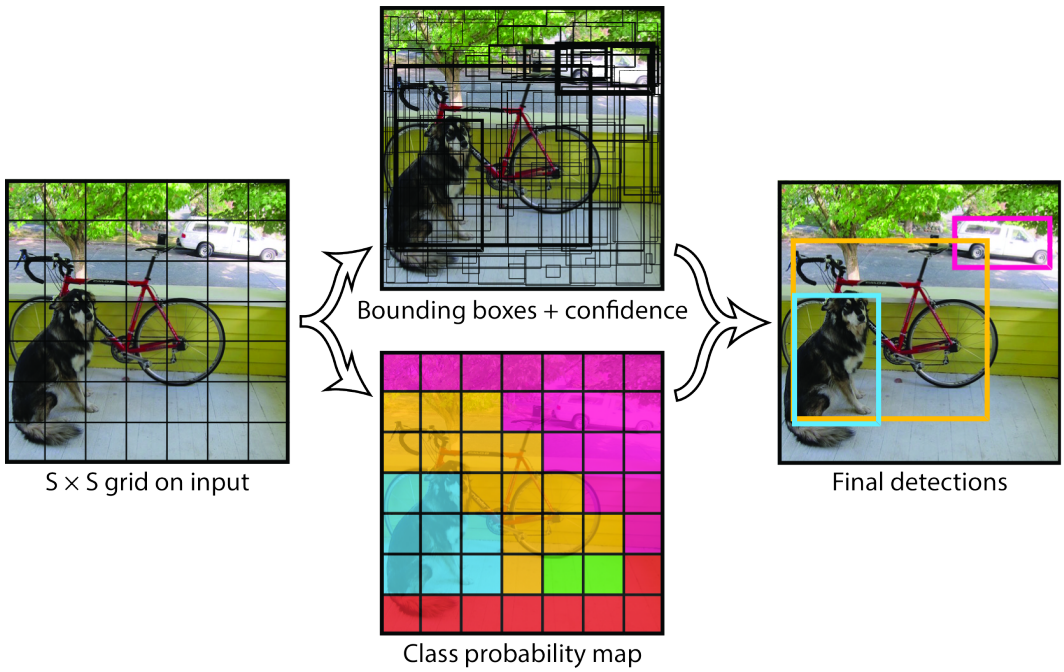


This shows which pixels activated the most for the predicted class. We can also get the salience mask for the "stone wall" class, which was not predicted:

YOLO

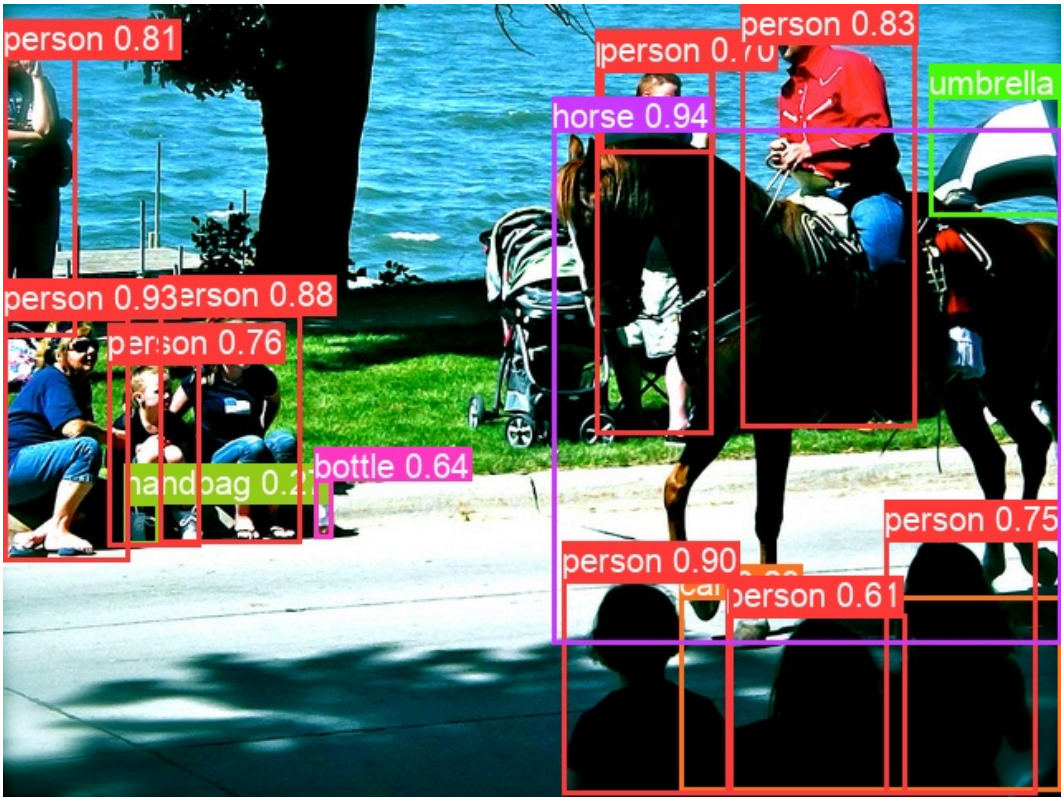
Salience is a useful debugging tool, but it is not precise enough for practical use. You Only Look Once (YOLO) [13] is a network which predicts both bounding boxes and class labels for each object instance in an image. It does this by dividing the image into a grid of cells, and for each cell predicting a set of bounding boxes and a class label for each box. The network is trained using a loss function that penalises incorrect predictions of both the bounding boxes and the class labels.

The architecture uses a simplified version of ResNet as its backbone and then directly regresses to the bounding boxes and class labels. The loss function is a combination of the sum of squared errors for the bounding boxes and the cross-entropy loss for the class labels. Here is an example of YOLO running on an image:



YOLO, cont.

YOLO is so fast that it can be run in real time on a GPU, even on mobile devices. Here is an example of the results of running YOLO on the COCO example image from earlier:



The number provided with each prediction is YOLO's confidence in that bounding box and class label. Note how it is able to handle partially occluded objects, multiple instances of the same class (including people from behind), and small objects like the water bottle.

Recognition

We have seen how we can use a CNN to detect distinct instances of a class in an image, but sometimes what we need is to identify whether a particular instance is one we have seen before.

Above, we see a group photo including a large number of people. One useful task would be to automatically label known people in the image. We can slide a window over the image, and detect people in the window, like below:

However, we need to use a different network to find people we have seen before. This task is called *recognition*.

LFW Dataset

The problem we have just described is an example of face recognition, where we want to detect that there is a face in the image, and then recognise which person it is. This is a much harder task than detection, as it requires the network to learn to recognise the same person's face in different poses, lighting conditions, and so on, not to mention appearance changes like makeup and facial hair.

The Labelled Faces in the Wild (LFW) dataset [3] contains 13K images of faces collected from the internet. Each face has been labelled with the name of the person pictured. Here are some examples:

We can use this dataset to train a CNN to perform face recognition.

Feature Embedding

When we trained a variational auto encoder, we learned a latent space embedding for images. We can do the same thing for faces, and then use this embedding to perform face recognition, which is the key idea behind FaceNet [15]. Specifically, we want to learn a latent space embedding such that the distance between two images of the same person is small, and the distance between two images of different people is large. We can do this by introducing a new loss, called the *triplet loss*:

$$\mathcal{L} = \sum_{i=1}^{N} \left[\left\| \mathbf{z}_a^i - \mathbf{z}_p^i \right\|_2^2 - \left\| \mathbf{z}_a^i - \mathbf{z}_n^i \right\|_2^2 + \alpha \right]_+$$

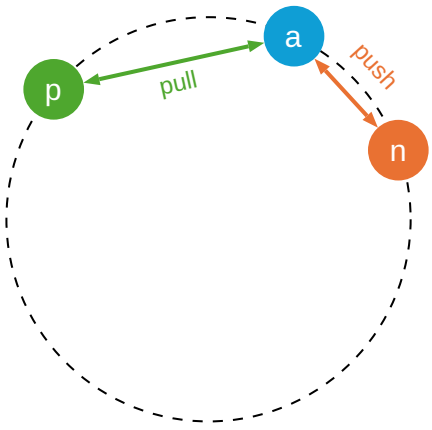
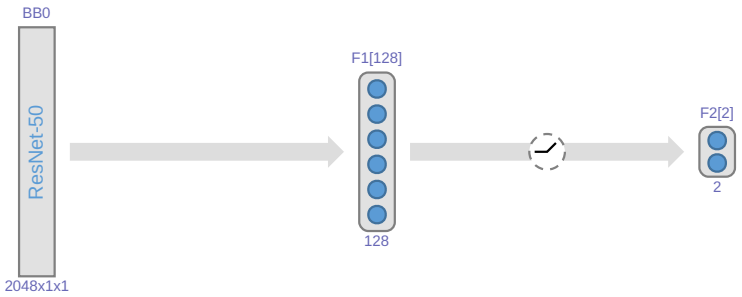


Figure 4: Triplet Loss

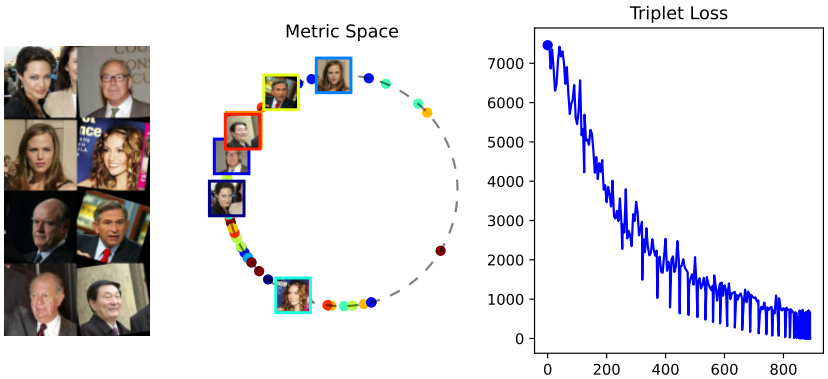
(NB the $[\,]_+$ notation means that we only apply the loss if the value inside the brackets is positive). One final element is that the learned embedding is constrained to have unit length. We construct triplets, like the one seen to the left, with an anchor point (a), a positive point (p), and a negative point (n). When the negative point is closer to the anchor than the positive point, we want to push it farther away while pulling the positive point closer. Face recognition can then be performed by embedding an image and computing its cosine difference to other face embeddings.

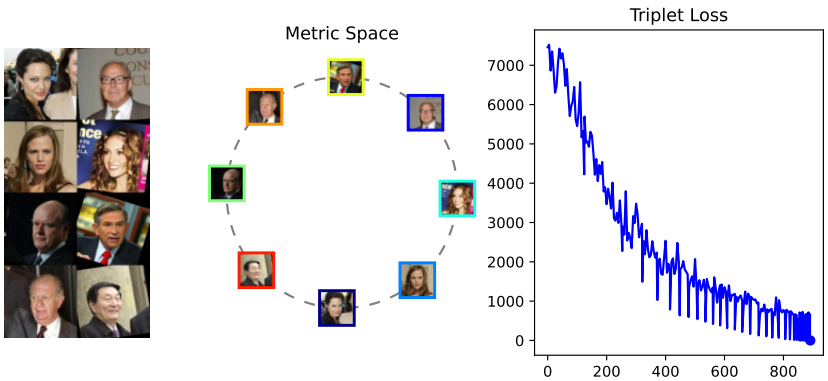
FaceNet

Here is a CNN architecture which uses ResNet-50 as its backbone and learns how to embed its final features into a 2D latent space:



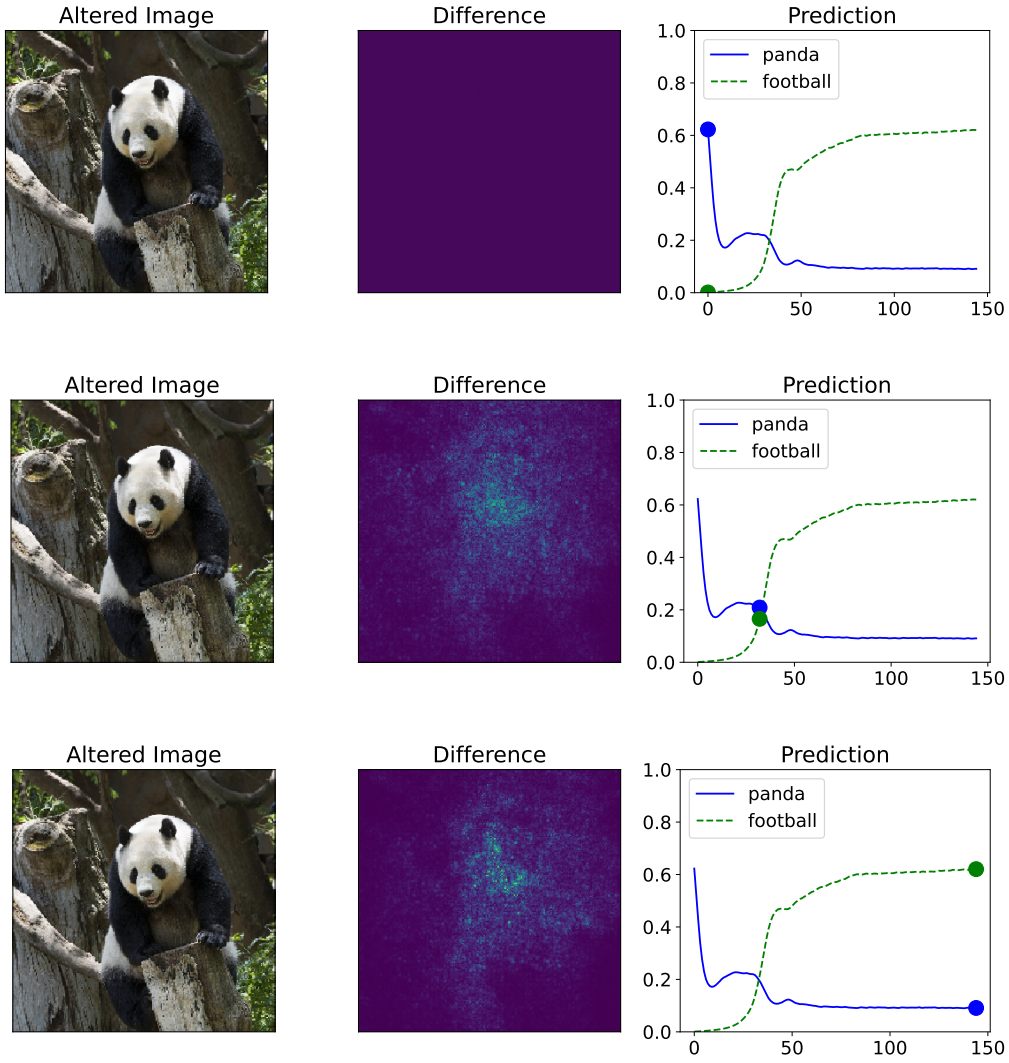
Here we can see the results of training this network on a dataset of 8 faces. Note how the loss is very noisy due to the selection of incorrect triplets changing from batch to batch.





Adversarial Attacks

We have looked at a few examples of how we can use back-propagation to manipulate inputs so that they produce unexpected outputs. This is called an *adversarial attack*, and it is a serious concern to keep in mind when building a system that incorporates a DNN as a component. In particular, as inputs grow in size, the distance to the nearest decision hyperplane decreases (due to the curse of dimensionality). For example, let us see what changes are needed to convince a state-of-the-art image classifier (ResNet-50) that a picture of a panda is actually a football:



Adversarial Attacks, cont.

As you can see, the images are visually identical, but the network is now as certain it is a football as it was previously that it was a panda. More worrying still, these principles can be applied in the real world. In 2017, researchers demonstrated that they could use stickers to fool a DNN into thinking a stop sign was a 40mph speed limit sign from a wide variety of distances, angles, and lighting conditions [1]:

Learned Noise

Stickers [40mph sign]

This cautionary note aside, it is an exciting time in which we are discovering many new uses and applications for DNNs. We should be aware of their limitations, understand how they work, and engineer systems which interact with them accordingly.

References

References

[1] K. I. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[3] Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.

[4] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

[5] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations*, 2015.

[6] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In *Proceedings of the 2nd International Conference on Learning Representations*, 2013.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, 2012.

[8] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2013.

[9] T-Y. Lin, M Maire, S Belongie, L Bourdev, R Girshick, J Hays, P Perona, D Ramanan, C. L. Zitnick, and P Dollár. Microsoft coco: Common objects in context. In *Proceedings of the European Conference on Computer Vision*, 2014.

[10] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(4):640–651, 2017.

[11] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[12] Mikel Olazaran. A sociological study of the official history of the perceptrons controversy. *Social Studies of Science*, 26(3):611–659, 1996.

[13] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.

[14] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[15] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[16] Paul Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.