Machine Learning

Xander Byrne

Lent 2023

1 Fondamentali

Machine learning is the development of models with many parameters $\boldsymbol{\phi}$ to achieve a task requiring intelligence. The achievement of this task is quantified by a loss function $L(\boldsymbol{\phi})$ which depends on the parameters, as well as on some training data. Training a machine learning model consists of using that training data to find parameters that minimise the loss function; the resulting model should ideally be able to generalise beyond the data that it was trained on.

For this preliminary discussion, it will be instructive to consider the case of *supervised learning*. Supervised learning seeks to train a model $\mathbf{f}(\mathbf{x}; \boldsymbol{\phi})$ to map inputs $\mathbf{x}$ to outputs $\mathbf{y}$.

1.1 Loss functions L

The loss function motivates the neural network; the training process proceeds solely by trying to find parameters that minimise it, when applied over training data $\{\mathbf{x}_i, \mathbf{y}_i\}$. In a supervised learning framework, the loss function quantifies the mismatch between the model predictions $\mathbf{f}(\mathbf{x}_i; \boldsymbol{\phi})$ and the targets $\mathbf{y}_i$:

$$L(\boldsymbol{\phi}; \{\mathbf{x}_i, \mathbf{y}_i\}) = \sum_i \|\mathbf{f}(\mathbf{x}_i; \boldsymbol{\phi}) - \mathbf{y}_i\|$$

Where $\|\cdot\|$ is some distance measure. For example:

- If y_i are scalars, this measure might simply be the square.
- If $\mathbf{y}_i$ are vectors (or a collection of numbers which might be written as a vector), then this might be the square Euclidean norm.
- If y_i are either 0 or 1 (i.e. we are doing a binary classification), we might use the *binary cross-entropy*:

$$L(\boldsymbol{\phi}; \{\mathbf{x}_i, y_i\}) = -\sum_i \left[y_i \ln \mathbf{f}(\mathbf{x}; \boldsymbol{\phi}) + (1 - y_i) \ln \left(1 - \mathbf{f}(\mathbf{x}; \boldsymbol{\phi})\right)\right]$$

Importantly, L should be a scalar, such that minimising it is meaningful.

If an output $\mathbf{y}$ is probabilistic with some distribution (e.g. a person's height), then ML models can be used to predict parameters of that distribution, rather than make a guess at the output. For example, we may want to find the probability distribution of $\mathbf{y}$ = people's heights parametrised by $\mathbf{x}$ = age. Now this probability distribution is usually a normal distribution,

with a μ and σ^2 which depend on age. In such cases, we could use an ML model to predict not simply the height of a person with a given age (which will probably be a bad prediction as heights have a distribution), but the mean and standard deviation of the distribution given that age: $(\mu, \sigma^2) = \mathbf{f}(\mathbf{x}; \boldsymbol{\phi})$. The loss function could then be given by the likelihood:

$$L(\boldsymbol{\phi}; \{\mathbf{x}_i, \mathbf{y}_i\}) = -\prod_i P[\mathbf{y}_i | \mathbf{f}(\mathbf{x}_i; \boldsymbol{\phi})] = -\prod_i \frac{1}{\sqrt{2\pi\sigma^2(\mathbf{x}; \boldsymbol{\phi})}} \exp\left(-\frac{(\mathbf{y}_i - \mu(\mathbf{x}_i; \boldsymbol{\phi}))^2}{2\sigma^2(\mathbf{x}_i; \boldsymbol{\phi})}\right)$$

where the negative sign is there because we want to *maximise* the likelihood rather than minimise it. (In practice we would usually write the loss function in terms of the logarithm of the probability for numerical reasons). This is essentially using machine learning to do Bayesian inference.

1.2 Gradient Descent and Backpropagation

How do we find the optimal parameters $\boldsymbol{\phi}$ to minimise L? The natural way of doing this is to compute $\boldsymbol{\nabla}_{\boldsymbol{\phi}} L$, and move the parameters in the direction of decreasing loss in parameter space: $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \lambda \boldsymbol{\nabla}_{\boldsymbol{\phi}} L$, where $\lambda > 0$ is the *learning rate*. (The learning rate is an example of a *hyperparameter*, a parameter which is not trained, but rather set by the user to govern how the training proceeds.) This method of training is called *gradient descent*, intuitively trying to find the minimum of the loss function in parameter space.

The gradient $\boldsymbol{\nabla}_{\boldsymbol{\phi}} L$ could be calculated numerically, but modern ML implementations are usually just able to analytically differentiate the loss function and calculate $\boldsymbol{\nabla}_{\boldsymbol{\phi}} L$ directly. For deep networks with many layers, L depends on the parameters in every layer, and the values in in

1.2.1 Modifications to Gradient Descent

Loss surfaces often have many local minima, which simple gradient descent algorithms are susceptible to getting stuck in if they start in an unlucky part of parameter space. As such, introducing some stochasticity is usually desired. *Stochastic gradient descent* (SGD) incorporates this by using a loss function which only uses a subset $\mathcal{B}$ of the training data (known as a *batch*) at each training step:

$$\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \lambda \frac{1}{N} \sum_{i \in \mathcal{B}} \boldsymbol{\nabla}_{\boldsymbol{\phi}} L(\boldsymbol{\phi}; \mathbf{x}_i, \mathbf{y}_i)$$

This leads to the parameters not necessarily taking the most downhill path possible in the loss space, potentially allowing evasion of local minima. Also, because the overall training set may not be representative of the entire data space, the stochasticity of training only on a batch at a time turns out to increase generalisability to unseen data.

The intuition of trying to find the bottom of a hill motivates several modifications of simple gradient descent. For instance, for a given learning rate, it is likely that towards the end of training successive steps will jump back and forth over the minimum, rather than settling to the bottom. As such, one modification is a *learning rate schedule*, whereby the learning rate starts large (to quickly get somewhere near to the minimum) and reduces over time, gradually tuning the parameters more and more finely to allow them to smoothly fall to the bottom of the loss surface.

Another modification is to take intuition from physics and think about a ball rolling around on a surface. To avoid this, a *momentum* can be assigned to the gradient descent at each step, so that if the parameters shoot into a local minimum they will fly up out the other side, hopefully ending up in a global minimum eventually. The implementation of this might look something like:

$$\mathbf{m} \leftarrow \beta\mathbf{m} + (1-\beta)\boldsymbol{\nabla}_{\boldsymbol{\phi}} L; \qquad \boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \lambda\mathbf{m}$$

where the momentum hyperparameter β controls how much we want to incorporate the momentum: $\beta = 0$ gives simple gradient descent. Variations on this theme include *Nesterov accelerated momentum*, which moves in the direction of what the gradient *would* be if we moved in the direction dictated by SGD:

$$\mathbf{m} \leftarrow \beta\mathbf{m} + (1-\beta)\left.\boldsymbol{\nabla}_{\boldsymbol{\phi}} L\right|_{(\boldsymbol{\phi}-\lambda\mathbf{m})}$$

A more advanced, complex, and commonly-used variation is *adaptive movement estimation*, which adds explicit time-dependence. For the timestep $t \geq 0$ and a new hyperparameter γ,

$$\mathbf{m} \leftarrow \beta\mathbf{m} + (1-\beta)\boldsymbol{\nabla}_{\boldsymbol{\phi}} L \qquad v \leftarrow \gamma v + (1-\gamma)\|\boldsymbol{\nabla}_{\boldsymbol{\phi}} L\|^2$$

$$\tilde{\mathbf{m}}_t \leftarrow \frac{\mathbf{m}}{1-\beta^t} \qquad \tilde{v}_t \leftarrow \frac{v}{1-\gamma^t}$$

$$\Rightarrow \boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \lambda \frac{\tilde{\mathbf{m}}_t}{\sqrt{\tilde{v}_t} + \epsilon}$$

where ϵ is a small constant to aid convergence.

1.3 Performance Metrics

How do we evaluate the performance of a model? Importantly, we cannot evaluate the performance of a model based on its performance on data that it has been trained on: by the very nature of training, it should get monotonically better at this. After training for a long time, performance on unseen data usually gets worse, as the model begins to *overfit* to the training data, picking up only on the noise and specific features of the training data rather than on features we want it to learn in order to generalise to unseen data. As such, we typically split our dataset into a training set and a *validation set*, which the model never uses to train, but is used to see track how good the model is at generalising. Early on in training, the validation loss usually falls as the model learns the most salient features in the data (both training and validation), before starting to rise once the model begins to overfit to the training set. There is a sweet spot where the validation loss reaches a minimum, and the model achieves maximum generalisability (for a particular choice of hyperparameters).

As mentioned above, one way of quantifying the performance of a model on unseen data is to calculate the loss function. But depending on what exactly one is interested in, one might track different performance metrics (though still only on the validation set). For example, if one is training a classifying algorithm, one might track the top-1 or top-5 accuracy of the model on the validation set.

1.4 Regularisation

Regularisation encompasses ways of using prior knowledge about what kinds of things the network should be learning as a way to mitigate against overfitting.

1.4.1 Explicit Regularisation

Sometimes the way in which models overfit to data manifests as some parameters becoming very large, both positive and negative, in order to bend as close as possible to the training data. This is often seen when one tries to fit a high-order polynomial to a dataset: the coefficients can become very large in magnitude. A common approach to mitigate against this is explicit regularisation: adding a "regularisation term" to the loss function. For example:

$$L_{\text{reg}} = L + \mu g(\boldsymbol{\phi}); \qquad g(\boldsymbol{\phi}) = \|\boldsymbol{\phi}\|$$

where $\|\cdot\|$ might be the L2 norm (L2 regularisation) or the L1 norm (L1 regularisation). This will discourage models with large parameters, and the incorporation of this validation term is equivalent to a prior on our model that the model should be "simple", i.e. with small parameters (this is effectively an *Occam term*).

1.4.2 Implicit Regularisation

Implicit regularisation is the numerical fact that taking finite steps in parameter space turns out to lead to the parameters missing the global minimum of the loss function slightly. Finite-step gradient descent on a loss function L turns out to be equivalent to infinitesimal-step (i.e. idealised) gradient descent on a loss function $\tilde{L}$:

$$\tilde{L}(\boldsymbol{\phi}) = L(\boldsymbol{\phi}) + \frac{\lambda}{4}\|\boldsymbol{\nabla}_{\boldsymbol{\phi}} L\|^2$$

Naturally, this doesn't change the positions of the minima in parameter space, where $\boldsymbol{\nabla}_{\boldsymbol{\phi}} L = \mathbf{0}$ by definition. This regularisation is effectively built-in to the use of finite-step gradient descent.

In stochastic gradient descent, the same principle turns out to favour regions in the parameter space where the gradients are similar between different batches, leading to the parameters reaching values that are ideal for generalisability (rather than just where one batch of data happens to be well-modelled). This may be why SGD generalises better than just GD.

1.4.3 Other Regularisation Heuristics

Early Stopping. As mentioned in the previous section, the validation loss (which, recall, quantifies the generalisability of the model) typically decreases initially before increasing again. By choosing to stop after T epochs, if T is chosen well we can train the model only up to the point when it starts overfitting.

Ensembling. To mitigate against the fact that some models will end up in local minima of the loss space, we can simply train several models with different initialisations (leading to different final parameters) and take the average of their predictions. Alternatively, rather than different initialisations for the different models,we can use different training data, for example by bootstrapping.

Dropout. At each training step, dropout randomly zeros out a fraction of a model's intermediate values (in the language of neural networks, these are *neurons* or *hidden units*). This has the effect of preventing parameters from conspiring as described earlier to take very large magnitudes, positive and negative, in order to (over)fit the noise in the training data.

2 Multilayer Perceptrons (MLPs)

2.1 Shallow Neural Networks

The simplest type of machine learning model is a multilayer perceptron. In the simple case of an MLP which has scalar inputs and outputs, the model takes the form:

$$f(x;\boldsymbol{\phi}) = \phi_{00} + \phi_{0i} \cdot a(\phi_{i0} + \phi_{i1}x)$$

where we have used the summation convention over i, which runs from 1 to D, where D is the *width* of the *hidden layer*. The hidden layer is a set of D *neurons*, which take the values $h_i = a(\phi_{i0} + \phi_{i1}x)$. Finally, a is the *activation function*: if there were no activation (or if the activation were linear), then the result f would be achievable simply with the parameters ϕ_{00} and ϕ_{0i}; the ϕ_{ij} would be redundant. As such, the activation function must be something non-linear: a common choice is the ReLU function.

The use of activation functions enables MLPs to be *universal approximators*: with a wide enough hidden layer, $f(x;\boldsymbol{\phi})$ can approximate a function or arbitrary complexity. Using ReLU as an example, we see that a hidden neuron h_i will take a value of $\phi_{i0} + \phi_{i1}x$, unless this is negative in which case it will take the value 0. Thus it will be a linear function on one side of $-\phi_{i0}/\phi_{i1}$, and 0 on the other. This will lead to a discontinuity in the final function f, which is a linear combination of the hidden neurons. The function f will therefore have (at most) D joints, dividing its domain into $D+1$ different regions where f takes a different linear form (though it will still be continuous as the individual ReLUs are). As $D \to \infty$, the domain is divided into infinitely many infinitesimal regions, able to approximate any function.

2.1.1 Multiple Inputs

If the function has multiple inputs $\mathbf{x} = \{x_1, x_2, \ldots, x_M\}$, i.e. the input space is M-dimensional, the hidden layer simply takes the extended form:

$$f(\mathbf{x};\boldsymbol{\phi}) = \phi_{00} + \phi_{0i} \cdot a(\phi_{i0} + \phi_{ij}x_j)$$

where the summation is now over j as well as i. The joints are now hyperplanes in the M-dimensional input space; there are D such planes. It can be shown that if $D \leq M$, then this generally divides the input space into 2^D regions. Within each of these regions, f takes a different linear form, and again, with high enough D a general scalar function can be approximated.

2.1.2 Multiple Outputs

Suppose x is a scalar, but there are multiple outputs $\mathbf{y} = \{y_1, y_2, \ldots, y_Q\}$. In that case, $\mathbf{f}$ must be a vector-valued function, with components f_k given by:

$$f_k(x;\boldsymbol{\phi}) = \phi_{k00} + \phi_{k0i} \cdot a(\phi_{i0} + \phi_{i1}x)$$

In this case, the real line is again partitioned into (generally) $D+1$ regions, and each of the components f_k have D joints; these will be at the same values of x_i for each component: $-\phi_{i0}/\phi_{i1}$.

2.1.3 Multiple Inputs and Outputs

For a network with D hidden units, operating on an M-dimensional input space and giving Q-dimensional outputs, the functional form of $\mathbf{f}$ is generally:

$$f_k(\mathbf{x};\boldsymbol{\phi}) = \phi_{k00} + \phi_{k0i}a(\phi_{i0} + \phi_{ij}x_j)$$

where i runs from 1 to D, j runs from 1 to M, and k runs from 1 to Q.

How many parameters are required in general? Each hidden layer $i \in [1, D]$ requires $M+1$ parameters, giving $D(M+1)$ parameters up to the hidden layers. Now each component of the output has parameters ϕ_{k00} and ϕ_{k0i}, giving $Q(D+1)$ further parameters, giving $Q(D+1) + D(M+1) = D(M+Q+1) + Q$ parameters in total.

2.2 Deep Neural Networks

Networks with one hidden layer are universal approximators, but for functions with significant second derivatives and large numbers of dimensions, one would need an impractically large number of parameters to get close. Deep networks allow a larger number of linear regions/joints for the same number of parameters, and are hence much more widely used.

Consider a network with two layers of widths D_1 and D_2, taking a one-dimensional input x and yielding a one-dimensional output y. The first hidden layer linearly scales the input and applies an activation, as before: $h_{1i} = a(\phi_{1i0} + \phi_{1i1}x)$, where i runs from 1 to D_1. The second hidden layer linearly scales the *first* hidden layer and applies an activation:

$$h_{2j} = a(\phi_{2j0} + \phi_{2ji}h_i) = a(\phi_{2j0} + \phi_{2ji}a(\phi_{1i0} + \phi_{1i1}x))$$

where j runs from 1 to D_2. We can then have a final layer with e.g. $f(x;\boldsymbol{\phi}) = \phi_{00} + \phi_{0j}h_{2j}$. This can be trivially extended to an input of any dimension, to give an output of any dimension. The above is also general to the choices of the number of hidden layers, and the widths of those layers, all of which are hyperparameters. Hyperparameters which define the structure of a neural network describe the network's *architecture*.

2.2.1 Backpropagation

Having described the functional form of a deep neural network, we would now like to learn how to train it: how can we change the parameters of the network to minimise a loss function? This involves taking the derivative of the loss function with respect to all of the parameters $\boldsymbol{\phi}$. We consider below the case with one input, one output, and two hidden layers:

$$\begin{aligned} h_{1i} &= a(\phi_{1i0} + \phi_{1i1}x) \\ h_{2j} &= a(\phi_{2j0} + \phi_{2ji}h_{1i}) \\ f &= \phi_{00} + \phi_{0j}h_{2j} \end{aligned}$$

For simplicity, consider first a batch size of 1, so that the loss function at each step is just $L(\boldsymbol{\phi}; x, y) = [f(x;\boldsymbol{\phi}) - y]^2$, where x is the input and y the target. To differentiate this with respect to the parameters, we will clearly have to apply the chain rule lots of times. To get the derivative with respect to the final bias ϕ_{00} right at the end, we will have:

$$\frac{\partial L}{\partial \phi_{00}} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial \phi_{00}}$$

which is not too bad because L depends on f very simply (merely quadratically) and $\partial f/\partial\phi_{00} = 1$. Similarly, to differentiate with respect to one of the ϕ_{0j}, we have $\partial L/\partial\phi_{0j} = [\partial L/\partial f][\partial f/\partial\phi_{0j}]$, the latter bracket being simply the corresponding h_{2j}.

As we go deeper back into parameters earlier in the neural network, we need more links in the chain (rule). For example, with respect to one of the ϕ_{1i1}, following the above sequence backwards we have:

$$\frac{\partial L}{\partial \phi_{1i1}} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial h_{2j}}\frac{\partial h_{2j}}{\partial h_{1i}}\frac{\partial h_{1i}}{\partial \phi_{1i1}}$$

Let's look at these derivatives in turn.

- $\partial L/\partial f$ has already been discussed; L is simply a quadratic function of f.
- $\partial f/\partial h_{2j}$ is simply equal to the parameter ϕ_{0j}. This derivative is in terms of another parameter; when evaluating it we use the original value of this parameter: we don't update any of the parameters until we know how they *all* should change.
- $\partial h_{2j}/\partial h_{1i}$ will depend on the derivative of the activation function:

  $$\frac{\partial h_{2j}}{\partial h_{1i}} = a'(\phi_{2j0} + \phi_{2ji}h_{1i}) \cdot \phi_{2ji}$$

  It is therefore important that activation functions are easy to differentiate.
- $\partial h_{1i}/\partial\phi_{1i1}$ also depends on the derivative of the activation function:

  $$\frac{\partial h_{1i}}{\partial \phi_{1i1}} = a'(\phi_{1i0} + \phi_{1i1}x) \cdot x$$

We see that as we evaluate the derivatives for parameters earlier and earlier on in the network, we need to first evaluate derivatives for later parameters. As such, when optimising deep neural networks, we typically work out the derivatives with respect to the later parameters first, storing them somewhere for when we need then for the derivatives wrt the earlier parameters. In other words, we start at the end of the network and work our way back to the start. This motivates the terminology of *backpropagation* to describe optimisation of deep neural networks.

2.2.2 Exploding, Vanishing, and Shattered Gradients

Increasing the depth of neural networks improves performance up to a point, but eventually training becomes volatile. If the weights of a network have a large variance, then the activations of a given layer will too. This will lead to large gradients, which will explode as we go to earlier layers in the backpropagation step. The opposite happens if the variance of the weights is very small. These are described respectively as the *exploding* and *vanishing gradients problem*. These can be mitigated by the *He initialisation*: rather than sampling the initial weights $\sim \mathcal{N}(0, 1)$, it turns out that the distribution $\mathcal{N}(0, 2/M)$, where M is the number of hidden input units, causes the variances of subsequent layers to be the same.

Alternatively, the gradients may end up changing very rapidly based on the input, the parameters charting a craggy loss surface. This phenomenon is known as the *shattered gradients problem*, and is the neural network equivalent of the butterfly effect. *Residual networks*, where the input to each layer is added back to the output can smoothen this out, but one then usually encounters exploding gradients. Within residual networks, this is mitigated using *batch normalisation*, whereby after the hidden units from a layer have been calculated, they are rescaled to learned values of the mean (γ) and standard deviation (δ), unique to each layer.

3 Convolutional Neural Networks (CNNs)

The MLP networks discussed in the previous section are also described as *fully-connected* networks, because each hidden unit depends on *all* of the hidden units in the previous layer. Images are very high-dimensional objects, and a fully-connected network with such a high-dimensional input would require an impractically large number of parameters. Fortunately, information in images is locally correlated: pixels in one area of the image are much more often related to nearby pixels than to pixels far away in the image. This can be exploited to design a more efficient type of neural network, a convolutional neural network, whose functional approximation power is focused on small patches at a time, achieving far better performance than a fully-connected network with the same number of parameters.

Consider a one-dimensional "image" $\mathbf{x} = (x_1, x_2, \dots)$. Consider also a *convolutional kernel* $\boldsymbol{\omega} = (\omega_1, \omega_2, \omega_3)$. The first layer of a convolutional network is computed by *convolving* the kernel with the input:

$$h_i = a(\omega_0 + \omega_1 x_{i-1} + \omega_2 x_i + \omega_3 x_{i+1})$$

To calculate the h_i we therefore simply slide the convolutional kernel across the image, and calculate the values of the hidden units thus.

The above naturally extends to a 2-dimensional image, where the kernel is often 3×3.

3.1 Variations on the Convolution Layer

3.1.1 Padding

There are edge effects to convolution that should be highlighted: what would h_1 be, if there is no x_0? One option is to not bother, and let there be no h_1, and similarly at the other end. This is called *valid* padding, and leads to the hidden layer being narrower than the input. Another option is to *zero-pad* the input, setting $x_0 = 0$ and similarly at the other end. In this way, if you want the next layer to have the same size as the input layer this can easily be done.

3.1.2 Stride and Max-Pooling

In the above, we slide the kernel along the image, moving one pixel at a time. We could also choose to slide the kernel along *two* pixels at a time. This is described as a *stride* of 2, and would halve the number of output units (give or take edge effects).

An alternative is to use *max-pooling* after each convolutional layer in a CNN. This involves taking a small (typically 2×2) window on the post-convolution image, and returning the maximum value of the (four) pixels. This is done such that the pooling windows are disjoint; the "downsampling" of the input is the same ($\pm$ edge effects) as 2-stride convolution.

So which is better, striding or max-pooling? Apparently, max-pooling typically works better for classification tasks, while striding makes more sense for emulation tasks.

3.1.3 Dilation

Another option is to use a *dilated* convolutional kernel. Such a kernel "skips" alternate pixels, giving a convolution of, for the 1D case:

$$h_i = a(\omega_0 + \omega_1 x_{i-2} + \omega_2 x_i + \omega_3 x_{i+2})$$

Dilated kernels can process a larger patch of the image without requiring more parameters.

3.2 Convolution Channels

Many images are RGB, and thus are described by $3 \times P \times P$ matrices, where P is the pixel-size of the image. We talk of the image having three *channels*. When processing multi-channel images with CNNs, the convolutional kernel has another dimension along the channel axis, in order to process this. So if we are applying a convolution of window size 5×5 to an RGB image, we will need the convolutional kernel to be of size $3 \times 5 \times 5$.

Applying a single $3 \times 5 \times 5$ kernel to an RGB image will result in a single-channel image. This is usually not very powerful, as with only one kernel there is only one *feature* that can be identified in an image. What is usually done, therefore, is to use a *series* of convolutional kernels, of the same dimension, and apply each of them to the same image. The output will hence also be, in a sense, multi-channel, with the number of channels being equal to the number of different convolutional kernels applied to the image. For the general case, where we are applying a $P \times P$ kernel to a C_i-channel image to give a C_o-channel output, the convolutional weights of a layer can be put a single matrix of dimension $C_i \times P \times P \times C_o$. (There will also be a bias for each of the output channels (the ω_0 above), which for a given layer can be stored in a C_o-dimensional vector.)

4 Transformers

Transformers were designed for natural language processing, which is difficult for computers. Words in a sentence are embedded as thousand-dimensional vectors, of which there may be hundreds in a passage of text: the input size is enormous, and is also of variable length. Also, when using natural language humans often use pronouns, which without context can be ambiguous: to assign meaning to what 'it' refers to in a sentence, a transformer must keep the whole passage in its "attention".

4.1 Tokenisation and Embedding

To numerically process a sentence, each word[1] must be mapped to a vector, called an *embedding* of that word. This is typically done by first assigning each word to an integer index, called a *token*; this might be simply the alphabetical position of that word in the dictionary. The tokens are then mapped to embeddings, using a matrix that is optimised during training.

4.2 Self-Attention Heads

The input to a transformer is a *sequence* of N embedding vectors $\{\mathbf{x}_m\}$. A *self-attention block*, also called a *head*, processes these, returning the same number of vectors of the same dimension.

For each input embedding vector, a *value* vector $\mathbf{v}_m$ is first computed, usually of the same dimensionality of the embedding vectors[2]; this is done in the usual way by linearly transforming the embedding vectors with a trainable weight matrix and bias vector: $\mathbf{v}_m = \boldsymbol{\beta}_\mathbf{v} + \boldsymbol{\Omega}_\mathbf{v}\mathbf{x}_m$.

The output of a self-attention layer is a sequence of vectors, one for each input word. The nth output vector is a weighted average of all the value vectors, where the weights depend on the two embedding vectors:

$$\mathbf{sa}_n(\{\mathbf{x}_m\}) = \sum_{i=1}^{N} A(\mathbf{x}_i, \mathbf{x}_n|\{\mathbf{x}_m\})\,\mathbf{v}_i$$

where $A(\mathbf{x}_i, \mathbf{x}_n)$ are attention weightings. These are described below.

4.2.1 Attention weightings

The attention weightings are between 0 and 1, and sum to one. $A(\mathbf{x}_i, \mathbf{x}_n)$ roughly corresponds to the importance of the ith word in the sentence to the understanding of the nth word. So if the nth word is "it", then the largest weighting would be to whichever word in the sentence "it" corresponds to; if this is the jth word, then $A(\mathbf{x}_j, \mathbf{x}_n)$ would be close to 1, and larger than other $Ah(\mathbf{x}_i, \mathbf{x}_n)$.

The attention weightings are calculated by a sub-network. Firstly, the embedding vectors are converted into two further sets of vectors: *keys* and *queries*:

$$\mathbf{k}_m = \boldsymbol{\beta}_\mathbf{k} + \boldsymbol{\Omega}_\mathbf{k}\mathbf{x}_m; \qquad \mathbf{q}_m = \boldsymbol{\beta}_\mathbf{q} + \boldsymbol{\Omega}_\mathbf{q}\mathbf{x}_m$$

[1] In reality, subwords are also processed, allowing for the learning of grammar as well as handling typos. Other sentence components, such as punctuation, bullet points, <end> etc. are also assigned a vector.

[2] This is in case one wants to create residual linkages in the network: for such linkages the input and output dimensions must be the same

Although similar to the calculation of the value vectors, the $\mathbf{k}$ and $\mathbf{q}$ vectors need not be of the same dimensionality as the embedding vectors, though they do need to be of the same dimensionality as each other. The attention weightings are then calculated as the softmax of the dot products between the keys and queries:

$$A(\mathbf{x}_i, \mathbf{x}_n | \{\mathbf{x}_m\}) = \frac{\exp\left(\mathbf{k}_i \cdot \mathbf{q}_n\right)}{\sum_j \exp(\mathbf{k}_j \cdot \mathbf{q}_n)}$$

$$\Rightarrow \qquad \mathbf{sa}_n(\{\mathbf{x}_m\}) = \sum_{i=1}^{N} \frac{\exp\left(\mathbf{k}_i \cdot \mathbf{q}_n\right)}{\sum_j \exp(\mathbf{k}_j \cdot \mathbf{q}_n)} \mathbf{v}_i$$

The self-attention head thus outputs another sequence of N vectors, of the same size and number as the input. These vectors are then each passed through a simple MLP network (one at a time) and the outputs for each vector are added together to give the final output of the transformer. *Layer normalisation* is often applied after the self-attention head and after the MLP, where the activations of the hidden units are renormalised to a trained mean/std.

In a typical transformer, there are several self-attention heads, each with their own value/key/query biases/weights. These effectively form multiple channels which are input to the MLP.

4.3 Positional Encoding

The above processing is invariant to the order of the words in the sentence: the sequence $\{\mathbf{x}_m\}$ could be in any order and give the same result. This is not ideal: the position of words in the sentence is crucial for its meaning.

The simplest way of incorporating the position of a word in a sentence for the transformer is to add another component to the embedding, and just put the sentence position in this final component, thus "tagging" the embedding with its position. A classier way of doing this is to *add* a positional encoding vector to each embedding vector, thus importing the positional information to subsequent layers.

5 Variational Autoencoders (VAEs)

Probabilistic generative models, a type of unsupervised ML model, attempt to model the probability distribution of the data, $f(\mathbf{x}|\boldsymbol{\phi}) \approx P(\mathbf{x})$. Such models can be trained in a maximum-likelihood sense, by minimising the loss function:

$$L(\boldsymbol{\phi}; \{\mathbf{x}_i\}) = -\sum_i \log f(\mathbf{x}_i|\boldsymbol{\phi})$$

If this distribution is well-modelled, then new instances can be sampled: the model is *generative*.

Latent variable models take an indirect approach to modelling $P(\mathbf{x})$: modelling the *joint* distribution $P(\mathbf{x}, \mathbf{z})$ of $\mathbf{x}$ and another variable $\mathbf{z}$, called the *latent variable*. The original distribution can then be recovered by integrating:

$$P(\mathbf{x}) = \int P(\mathbf{x}, \mathbf{z})\, \mathrm{d}\mathbf{z} = \int P(\mathbf{x}|\mathbf{z})P(\mathbf{z})\, \mathrm{d}\mathbf{z}$$

At first glance this seems to have made things more complicated, but it enables very complex distributions $P(\mathbf{x})$ to be modelled by breaking it down into the convolution of two other distributions which can be much simpler.

For example, we might choose $P(\mathbf{z})$ to be a standard multivariate normal distribution; we might choose $P(\mathbf{x}|\mathbf{z})$ to be a normal distribution with mean $\mathbf{f}_\mu(\mathbf{z}|\boldsymbol{\phi})$ and covariance $\mathbf{f}_\Sigma(\mathbf{z}|\boldsymbol{\phi})$, each generated by a neural network. In this way, the model distribution for $P(\mathbf{x})$ is a shmear of Gaussians. New examples $\mathbf{x}^*$ can be generated by first generating $\mathbf{z}^*$ from the standard normal, then calculating $\mathbf{f}_\mu(\mathbf{z}^*|\boldsymbol{\phi})$ and $\mathbf{f}_\Sigma(\mathbf{z}^*|\boldsymbol{\phi})$, and finally generating $\mathbf{x}^*$ from the normal distribution with that mean and covariance.

I don't really know what else is needed here; the course sort of trailed off at this point, but the examinable material calls for a "conceptual understanding of VAEs". I'll give it a go.

VAEs consist of an encoder and a decoder, to map data $\mathbf{x}$ to and from their latent space representations $\mathbf{z}$. The encoder consists of approximating the posterior distribution $q(\mathbf{z}|\mathbf{x})$, which is taken to be a normal distribution with mean $\mathbf{g}_\mu(\mathbf{x}|\boldsymbol{\theta})$ and covariance $\mathbf{g}_\Sigma(\mathbf{x}|\boldsymbol{\theta})$. After encoding a data instance $\mathbf{x}$, a latent sample $\mathbf{z}^*$ is sampled according to this posterior distribution $q(\mathbf{z}|\mathbf{x})$. Finally, the decoder directly predicts the data $\mathbf{x}$, using a second neural network: $\mathbf{x}^* = \mathbf{f}(\mathbf{z}^*|\boldsymbol{\phi})$. The probability of obtaining that data point $\mathbf{x}^*$ is estimated; I'm not really sure how. The model is then trained to maximise the probability of the data but also to massage the latent distribution $q(\mathbf{z}|\mathbf{x})$ into a form we might want, such as a standard normal distribution, in order to give the latent space useful structure.

6 Diffusion Models

Diffusion models are another kind of probabilistic generative model. They are trained to reverse a process of multistage stochastic degradation of an image, typically the progressive addition of random noise. Once trained, one can give the model some random noise, and it will "denoise" it into a recognisable image.

6.1 Diffusion Process

Diffusion of an image $\mathbf{x}$ proceeds by progressively mixing the image with Gaussian noise $\boldsymbol{\epsilon}_t$:

$$\mathbf{z}_t = \sqrt{1-\beta_t}\,\mathbf{z}_{t-1} + \sqrt{\beta_t}\,\boldsymbol{\epsilon}_t; \qquad \mathbf{z}_0 \equiv \mathbf{x} \qquad \Rightarrow \qquad \mathbf{z}_t|\mathbf{z}_{t-1} \sim \mathcal{N}\Big(\sqrt{1-\beta_t}\,\mathbf{z}_{t-1}, \beta_t\mathbf{1}\Big)$$

The *noise schedule* β_t controls how much noise is added at each diffusion step. After a large number T steps, the resulting image $\mathbf{z}_T$ will essentially be itself random noise.

Usefully, noising with this prescription means that there is an analytic distribution for $\mathbf{z}_t|\mathbf{x}$. For example, to get $\mathbf{z}_2$ in terms of $\mathbf{x}$,

$$\begin{aligned}\mathbf{z}_2 &= \sqrt{1-\beta_2}\left(\sqrt{1-\beta_1}\,\mathbf{x} + \sqrt{\beta_1}\,\boldsymbol{\epsilon}_1\right) + \sqrt{\beta_2}\,\boldsymbol{\epsilon}_2 \\ &= \sqrt{(1-\beta_2)(1-\beta_1)}\,\mathbf{x} + \sqrt{\beta_1(1-\beta_2)}\,\boldsymbol{\epsilon}_1 + \sqrt{\beta_2}\,\boldsymbol{\epsilon}_2\end{aligned}$$

Now $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$, so the two final terms $\sim \mathcal{N}(\mathbf{0}, \beta_1(1-\beta_2)\mathbf{1})$ and $\sim \mathcal{N}(\mathbf{0}, \beta_2\mathbf{1})$ respectively. Adding them together gives more Gaussian noise, with the variances added:

$$\begin{aligned}\mathbf{z}_2 &= \sqrt{(1-\beta_2)(1-\beta_1)}\,\mathbf{x} + \sqrt{1-(1-\beta_1)(1-\beta_2)}\,\boldsymbol{\epsilon} \\ &\equiv \sqrt{\alpha_2}\,\mathbf{x} + \sqrt{1-\alpha_2}\,\boldsymbol{\epsilon}\end{aligned}$$

where $\alpha_t \equiv \prod_{i=1}^{t}(1-\beta_i)$. By induction we can extend this to any t, allowing us to write:

$$\mathbf{z}_t|\mathbf{x} \sim \mathcal{N}[\sqrt{\alpha_t}\,\mathbf{x}, (1-\alpha_t)\mathbf{1}]$$

Thus we have derived the distribution of $\mathbf{z}_t$, for any t in the diffusion process.

6.2 Training

Diffusion models predict the (cumulative) noise that has been added to an image. During training, an integer t is sampled randomly from the range $[1, T]$, and an image is noised by an amount according to the noise schedule[3]. The network is then trained to take in this noised image as input, and return as output the noise that was added: for a single image $\mathbf{x}$,

$$\ell = \left\|\mathbf{f}\left(\sqrt{\alpha_t}\,\mathbf{x} + \sqrt{1-\alpha_t}\,\boldsymbol{\epsilon}, ; t, \boldsymbol{\phi}\right) - \boldsymbol{\epsilon}\right\|^2$$

One might ask why we don't just train the network to predict the cumulative noise added by the timestep T? Why bother learning how to predict noise that was added by the time of each intermediate step? The answer is that neural networks learn small steps more easily than large steps, so by learning how much noise is added along the way, the network has an easier task: simply put all the small steps together.

[3]This could be done by repeatedly noising the image (using the β_t), to update $\mathbf{z}_t$ to $\mathbf{z}_{t+1}$, but since a way of going directly from the image $\mathbf{x}$ to $\mathbf{z}_t$ was derived above (using the α_t), we just do that instead, which saves time.

6.3 Sampling

Once trained, we have a function $\mathbf{f}$ that can take in a partially-noised image and return the noise that might have been added to the original image. We can use this to generate some realistic-looking images from pure Gaussian noise. After T timesteps, the image will have essentially become random noise, so if we can start with an image of random noise $\mathbf{z}_T^*$ and denoise it, then we will end up with a realistic image.

Again, we could try to go directly from $\mathbf{z}_T^*$ to $\mathbf{x}^*$, but performance turns out to be better if we go from $\mathbf{z}_T^*$ to $\mathbf{z}_{T-1}^*$, progressively down until $\mathbf{z}_1^*$, which we denoise to $\mathbf{x}^*$. It turns out that the way to do this is as follows:

$$\mathbf{z}_{t-1}^* = \frac{1}{\sqrt{1-\beta_t}}\mathbf{z}_t^* - \frac{\beta_t}{\sqrt{1-\alpha_t}\sqrt{1-\beta_t}}\mathbf{f}(\mathbf{z}_t^*; t, \boldsymbol{\phi}) + \sqrt{\beta_t}\,\boldsymbol{\epsilon}^*$$

where again $\boldsymbol{\epsilon}^* \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$. If I had more energy, or if it were taught better, I might have been motivated to find out where on earth this comes from. Finally, we denoise $\mathbf{z}_1^*$ using:

$$\mathbf{x}^* = \frac{1}{\sqrt{1-\beta_t}}\mathbf{z}_1^* - \frac{\beta_1}{\sqrt{1-\alpha_1}\sqrt{1-\beta_1}}\mathbf{f}(\mathbf{z}_1^*; t=1, \boldsymbol{\phi})$$

6.4 Time Encoding

Note that above the neural network takes the timestep t as an input. As with the discussion of positional encoding in the section on transformers, the timestep can be implicitly included with the input, perhaps by appending it on the end, or adding an encoding vector.