Visualising the Hidden Neurons[1] of a Non-convolutional DNN Trained on the MNIST Database

Ben Auer

Introduction:

The question being investigated in this project was as follows:
What roles do specific nodes in the hidden layers of a non-convolutional image classifier Deep Neural Network (DNN) develop after training on the Modified National Institute of Standard and Technology (MNIST) database?

It was hypothesized that some rudimentary versions of capabilities that are manually implemented in Deep Convolutional Neural Networks, such as curve or line detectors, would naturally emerge as capabilities of the non-convolutional DNN's hidden nodes, and would be evident when visualising the kind of inputs that those nodes respond to most significantly.

A DNN was trained to high performance on the MNIST test set using ReLU and SoftMax activation and cross-entropy loss, and then analysis of the DNN began. The method used for visualisation was optimisation of image inputs for activation of a target node in the network, by backpropagating the activation, and the method was validated by testing on output neurons (with known roles). Initially the images produced were mostly too noisy to be interpretable. Consequently, the optimiser was initialised from existing MNIST images in an attempt to narrow the search space, but it was found that this led to the generation of adversarial examples, which highly activated an incorrect output node despite appearing almost identical to the original image (before optimisation).

Following this, initialisation was reverted to near-zero values, but regularisation techniques were introduced to place stronger constraints on the optimised images – primarily in the form of penalising Total Variation (TV) and the L1 norm. TV regularisation had a drastic effect on the image quality and led to the emergence of smoother, more concentrated features, while L1 drove down unnecessary background pixels. In combination, these effects also resulted in far less noisy, and far more interpretable, visualisations.

It was also discovered here that adding more regularisation techniques was correlated with increasing failures in optimisation. Indeed, just TV and L1 regularisation had already narrowed the search space sufficiently that there was a strong convergence towards certain features in the visualisations, for both output and hidden nodes, and these patterns began to elucidate the functions of the neurons, and the differences between them. For instance, it seemed that nodes in the earlier and larger hidden layers were sensitive to pixels that were more widely distributed throughout the image space, while nodes in later and smaller hidden layers responded to more concentrated regions. When querying the DNN with these visualisations, those from the former category activated a range of different output nodes, while those from the latter category often correlated strongly with a single digit. Additionally, some nodes activated synergistically in small clusters, while other nodes were 'dead' and never activated at all.

[1] The words 'neuron' and 'node' are used interchangeably in this document and in the appended code.

The visualisations did not confirm the initial hypothesis as they suggested that, rather than learning to recognise very generalised geometric structures that appeared in many MNIST images, the hidden nodes had become specialised to spatial locations and were responding to precisely located features, i.e., regions of pixels, which gradually decreased in size as layers became larger, down to individual pixels in the earliest layers. This is quite different from the more holistic methods that humans use (cognitively) to assess images, and seemingly even quite different to the methods used by more powerful convolutional DNNs. But the ontology of the MNIST classifier is much simpler than those other neural networks since it is based only on quite a narrow style of images. And hence the roles developed by its hidden nodes, though unexpected, are well-adapted to its objectives.

DNN Configuration and Training:

The project was carried out in Python with a limited number of simple libraries, primarily NumPy, with some usage of SciPy, and the pyplot package from matplotlib. All calculations required for DNN training was implemented manually using array-like matrices in NumPy.

The DNN was initiated using Kaiming initialisation (anticipating ReLU activation) (He et al., 2015) with a 7-layer architecture, with 1250-1000-750-500-250-100-32-10 neurons (and 784-dimension input). The first 5 layers were modelled after Ciresan et al.'s (2010) 99.65%-performance-achieving non-convolutional MNIST classifier, but with exactly half the number of neurons due to more limited computational resources. Additionally, 2 more layers were added to further narrow the number of nodes and encourage stronger compression of image features by the DNN for more interpretable visualisation results – especially in the relatively sharp transition to the last hidden layer of 32 nodes (~ $^1/_3$ of the 6th layer's 100 nodes). The gradual narrowing of the first 5 layers allowed this sharper narrowing to be done downstream without sacrificing performance. It was also deduced that the lower size of the final layer would increase speed of certain computations for image optimisation and other visualisation methods – namely, summing the activations of all nodes in the layer.

The loss function used was the mean cross-entropy loss, with SoftMax activation for the final (output) layer. Cross-entropy loss is highly suitable for MNIST classification since each image has only one correct class, and the loss is the sum of the natural logarithm of the activation of the final layer node corresponding to the correct class for each image in a batch. (All other terms are automatically 0 since the one-hot target vector is multiplied elementwise with the final layer outputs.) As the correct node activation approaches 1 (i.e., 100% probability) the natural logarithm of that term approaches 0. The activation of all other output nodes for that image decreases simultaneously because SoftMax guarantees that the activations sum to 1, so as the DNN updates, it gradually places more and more probability weight on the correct classification. ReLU activation was used for the hidden layers since it is simple but can still achieve high performance on MNIST (e.g., Glorot, Bordes, & Bengio, 2011), and it was expected that a better-performing classifier would have more robust internal representations of the images.

The formulas for the ReLU and SoftMax activation function, the cross-entropy loss and the derivatives of all of these can be found in the Appendix, item 1.

The DNN was trained for 20 epochs on the MNIST training set with a batch size of 32 and a learning rate of 0.01, and a further 5 epochs at a learning rate of 0.001[2] (same batch size). The loss was measured by an exponential moving average with weighting 0.9999, which steadily decreased until the end of training, where it was approximately 0.01, and the DNN then achieved a performance of 98.06% correct classification (with argmax), i.e., 1.94% error rate. The weights and biases at this point were used for visualisation and other analysis of the DNN.

Visualisation by Image Optimisation:

A new class based on the DNN class was implemented to visualise the hidden neurons by optimisation of input images, which initiated a 784-element vector consisting either of near-zero values (0.01) or randomly distributed values and defined a training function. The function would optimise the image for activation of a particular target node in the DNN by updating the image pixel values based on the gradient of the node activation with respect to the input (i.e., backpropagating the activation), with a penalty term consisting of the gradient of the mean activation of all other nodes in the layer (times an adjustable weighting), to encourage unique activation of the target node.

As implied above, any neuron's activation when querying a DNN is differentiable with respect to the query input (image in this case), which enables this method. The activation of an entire layer is also differentiable with respect to an image input, and this layer gradient is equivalent to summing the gradients for all the individual nodes in a layer, which provides an efficient way of computing the gradient of the penalty term described above, to backpropagate the penalty as well.

Moreover, the partial derivative formulas follow a very similar pattern to the gradient normally used in backpropagation for updating the weights of a DNN during training (which is comprised largely of the partial derivative of the final layer node activations with respect to the weights), but they differ in the last factor, which is the weights for the first layer, rather than the inputs. The partial derivative for a single node also uses only the weights and biases relevant to that node. The full formulas are in the Appendix, item 2.

To validate the experimental method, the visualisation class was first tested by targeting the output neurons of the DNN. This is because the role of each output neuron was already known in advance and therefore the results could be evaluated against this knowledge, whereas for the hidden neurons there was no such frame of reference. When initialising from a near-zero vector, the image appeared to evolve almost deterministically with a given neuron and learning rate, and the optimiser was successful insofar as it achieved very high unique activation of the target node (approaching 1 rapidly and indefinitely). In fact, the optimiser was so effective that the learning rate could be increased by many orders of magnitude until the image reached over 99% activation within a few updates. It was also observed that the overall image contrast increased with the learning rate, and with very high

[2] The learning rates were lower than intended due to accidentally dividing the gradient by the number of output nodes (10), but performance was still as high as expected (likely due to extensive training length).

learning rates, most pixels would be either completely white or black, suggesting that the easiest paths for the optimiser required merely maximising or minimising the values of certain pixels.

Nevertheless, all these images had very high levels of noise, and were not interpretable, even with a modest learning rate (e.g., 0.1). There was also a further issue that arose during optimisation: pixel values being pushed below 0 or above 1 – i.e., outside of the normalised range of the training distribution. This would presumably have limited the range of conclusions that could be drawn from any visualisations, so a numpy.clip image function (and its derivative) was implemented in the optimiser to rectify this, which simply cuts off all values at a specific maximum and minimum, 0 and 1 in this case (formulas in Appendix, item 3). With clipping, the images had some darker or brighter regions which could plausibly have corresponded with the relevant digit (e.g., a hole near the centre of images optimised for the 0 node) but were still far too noisy to be appropriately called visualisations of the output neurons. The same occurred to an even greater extent when targeting hidden nodes, where the images generated had no decipherable meaning, despite also uniquely activating the target node.

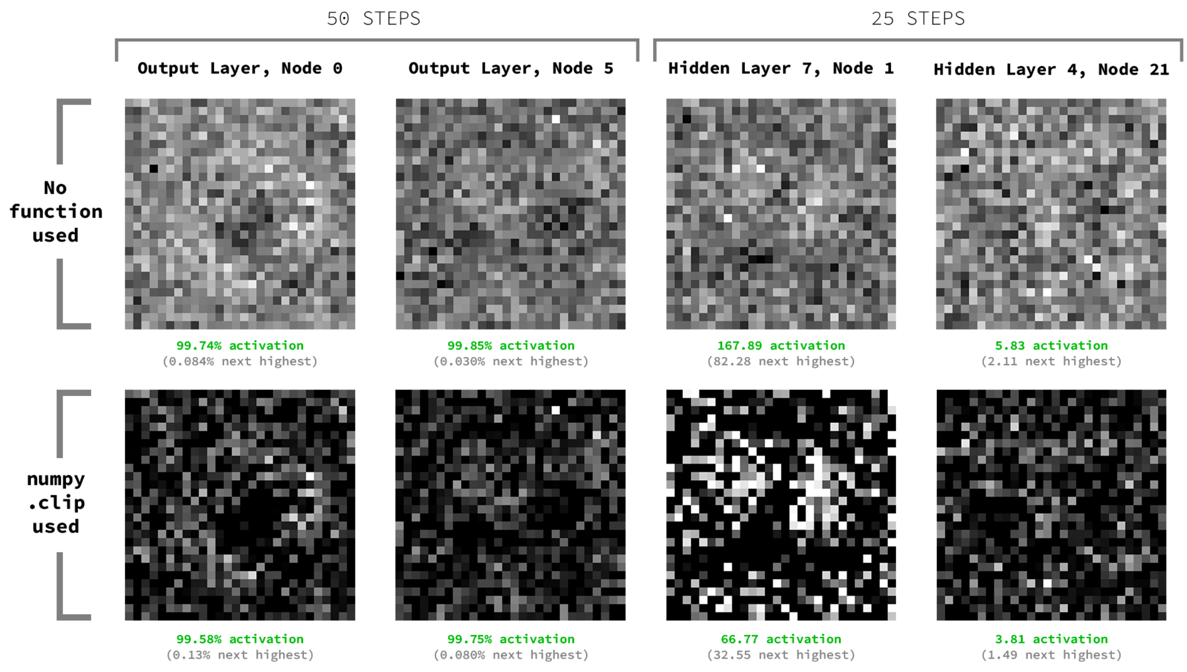

Figure 1. Images optimised for target nodes without any regularisation, above counterparts which differ only by the usage of a numpy.clip function during and after optimisation. The number of optimisation steps was held constant at 50 for the output layers and 25 for the hidden layers (due to their different activation functions). The clipped images have a smaller range of pixel values and therefore appear less noisy, despite no other regularisation being used. The activation of each image is written below it in green text, and the activation of the second-most activated node below this in grey text.
NB: Adobe Photoshop was used to scale up the images with Nearest Neighbour resampling, which preserves exact pixel sizes and colours from the original images.

Next, the function was modified to initialise from actual MNIST images, to create some likelihood of maintaining their features during optimisation. This technique was again tested first with the output neurons of the DNN, selecting a target node that was different to the correct classification of the image, which was (usually) already highly activated by the original image. It was hypothesized that the optimiser would convert the original image to a new MNIST-style drawing of the digit that would highly activate the target node. But contrary to expectations, the optimiser produced images strikingly similar to the originals, typically only with slightly more noise, that could still be easily identified by a human as representing the same digit but were classified by the DNN incorrectly with high confidence (over 0.99). And, of course, this meant that the sum of the other output node activations was less than 0.01, including the correct output node. It is also worth noting that this was with the numpy.clip activation function and, as anticipated, the pixel values for the images were all between 0 and 1, which meant that the optimiser was not simply exploiting numerical anomalies (like negative pixel values) but genuine visual features (like noise).

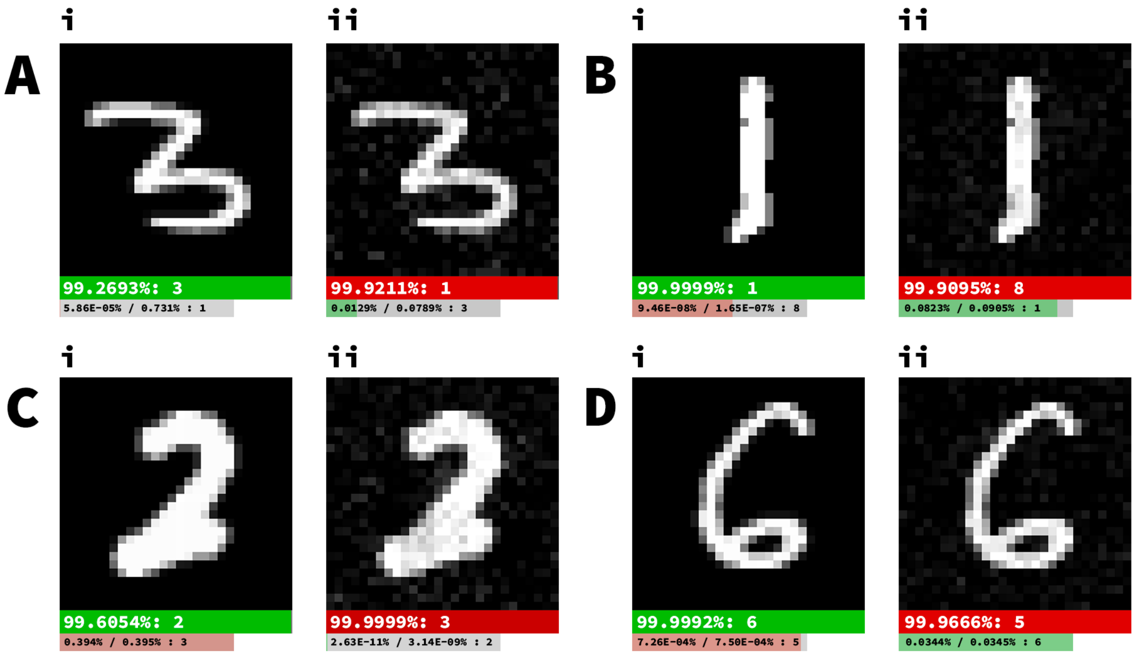

Figure 2. MNIST images alongside the adversarial examples generated from each of them by the visualisation optimiser. The first large bar below each image shows the activation of the most activated output node for that image (green for correct, red for incorrect), as a percentage, with the node number. The size of the bar also represents this – in green for the correct classification, and red for an incorrect classification. The second, lighter and smaller bar (not to scale) shows what proportion of the total activations from other nodes (in gray) come from the node corresponding to the other image in the pair (same colour scheme).
For example, the image in Figure 2.Ai activates node 3 the most (99.2693%) and activates node 1 5.86E-05% out of 0.731% total activations. Conversely, the adversarial example in Figure 2.Aii activates node 1 the most (99.9211%), and node 3 it activates 0.0129% out of 0.0789% total other activations.

In other words, the optimiser generated adversarial examples for the DNN when initialised from an actual MNIST image and was therefore not informative regarding the effective function of the target neurons within the context of the MNIST datasets. This also implied that any images generated using the same strategy with hidden neurons could be adversarial examples themselves, even if they were interpretable.

Furthermore, random initialisation had been included at first to account for possible difficulties with the optimiser increasing pixels from very low initial values, but this was not an issue with optimisation thus far, so from this point onwards near-zero initialisation was used as the predominant strategy. And indeed, this strategy continued to be successful and also resulted in less background noise than random initialisation.

It was clear that other regularisation techniques were needed to reduce both noise and adversarial examples, so Total Variation (TV) regularisation (Rudin, Osher, & Fatemi, 1992) and L1 (LASSO) regularisation (Tibshirani, 1996) were introduced. The usage of TV for feature visualisation was first introduced by Mahendran and Vedaldi (2015) – though in this case both techniques were inspired by Olah, Mordvintsev and Schubert (2017).

A discretised anisotropic (2D) version of TV was used on account of the discrete nature of the image and the relative simplicity of implementing this formulation, which is as follows, for an image u:

$$\text{"}[TV(u)] = \sum_{\substack{1\leq i \leq N \\ 1\leq j\leq N}} \left|u_{i+1,j} - u_{i,j}\right| + \sum_{\substack{1\leq i \leq N \\ 1\leq j\leq N}} \left|u_{i,j+1} - u_{i,j}\right| \text{"}$$

(Caselles, Chambolle, & Novaga, 2014, p. 1743).

Hence the derivative is as follows for any pixel $u_{a,b}$ (derivation in Appendix, item 4):

$$\frac{dTV}{du_{a,b}} = \operatorname{sgn}\left(u_{a,b} - u_{a-1,b}\right) - \operatorname{sgn}\left(u_{a+1,b} - u_{a,b}\right) + \operatorname{sgn}\left(u_{a,b} - u_{a,b-1}\right) - \operatorname{sgn}\left(u_{a,b+1} - u_{a,b}\right)$$

The derivative was implemented in the code by first creating a padded version of the image for efficient comparison of neighbouring pixels. (See the Appendix, item 5, for more detail.)

This TV formula essentially measures the absolute difference between variants of the image that are translated slightly along vertical or horizontal axes, thus encouraging optimisation towards smoother images, while the L1 norm simply measures the summed absolute value of all the pixels, thus discouraging extreme values. Note that both measures are minimised since their partial derivatives are subtracted from the overall optimisation gradient. The formulas for L1 regularisation are as follows:

$$\text{For } I = \begin{bmatrix} I_1 \\ \vdots \\ I_{784} \end{bmatrix}, L_1(I) = \sum\nolimits_{i=1}^{784} |I_i|\,, \text{so } L_1{}'(I) = \operatorname{sgn}(I)$$

where I is an input image.

Due to a serendipitous oversight in the initial implementation here, it was discovered that deliberately not incorporating the numpy.clip derivative into the TV and L1 regularisation gradients led to more smooth and less noisy images that were far more interpretable. In retrospect it seems this was because the numpy.clip derivative would set gradients to zero for all pixels beyond the minimum or maximum bounds, meaning those pixels would become trapped permanently at the bounds, and this would gradually affect more and more pixels as optimisation proceeded, thereby populating the image with extreme values (i.e., completely white or black pixels). But the regularisation gradients, when computed independently of this, could keep those pixels dynamic, and potentially even return them within the bounds. As such, the image could continue to develop further towards something that better matched the regularisation constraints, while optimisation as a whole ensured that the clipped values still highly activated the target node. It is also worth noting that the numpy.clip derivative is always 1 within the bounds, so there were no other effects of removing it from the regularisation gradients.

With this method, even TV regularisation alone had a drastic effect. When initialising from near-zeros or even from actual MNIST images with a sufficient TV regularisation weighting, image features were smooth and there was almost no irrelevant noise. And even with random initialisations the amount of noise was significantly less than earlier. The images also tended to converge on consistent patterns regardless of initialisation. When targeting output nodes, these patterns were very reminiscent of each digit, and somewhat like the patterns observed in the noisy images earlier. In fact, clear similarities were visible when lining up all these images alongside the average of all the MNIST training data for that digit, though some regions or segments of the digits were missing or emphasized in the optimised images, perhaps indicating what was uniquely important for classification of each specific digit. Weighting the TV regularisation at different levels (while holding the learning rate constant) also resulted in different styles of images, though the same underlying patterns were evident in all cases.

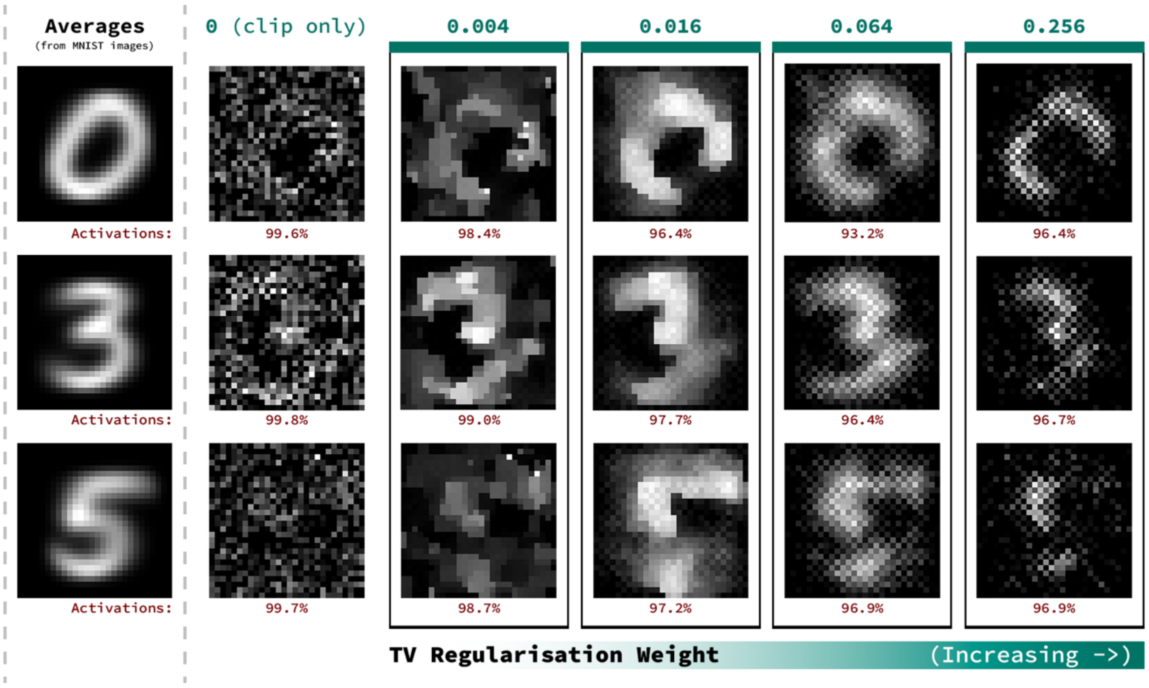

Figure 3. Images optimised for activation of a target output neuron (0, 3, or 5), alongside the average of all MNIST images which highly activate that node (left side). Each row corresponds to a node. TV regularisation weight increases exponentially towards the right side of the figure, starting from 0.004 and ending at 0.256. All images were initialised from near-zero values, and optimised for 100 steps (with clipping) to ensure high activation of the target node. The activation induced by each image is written below it in red.

L1 regularisation, on the other hand, seemed to work by driving down the values of any pixels that were unnecessary for activation of the target node. This resulted in more minimalistic versions of the images, with many more black pixels (at value 0), and smaller and more concentrated regions of white pixels. This type of effect was observed with or without TV regularisation also being used. The range of possible values for TV and L1 regularisation that worked in combination was limited, however. If both were too strong, the optimiser would fail to reach high activation of the target node, while if L1 regularisation was too strong, all images would be reduced to only a handful of (non-black) pixels. But a moderate amount of TV regularisation with a small amount of L1 regularisation (e.g., 10% of the TV weight) did yield highly-activating images that seemed cleaner and more precise than with TV regularisation alone.

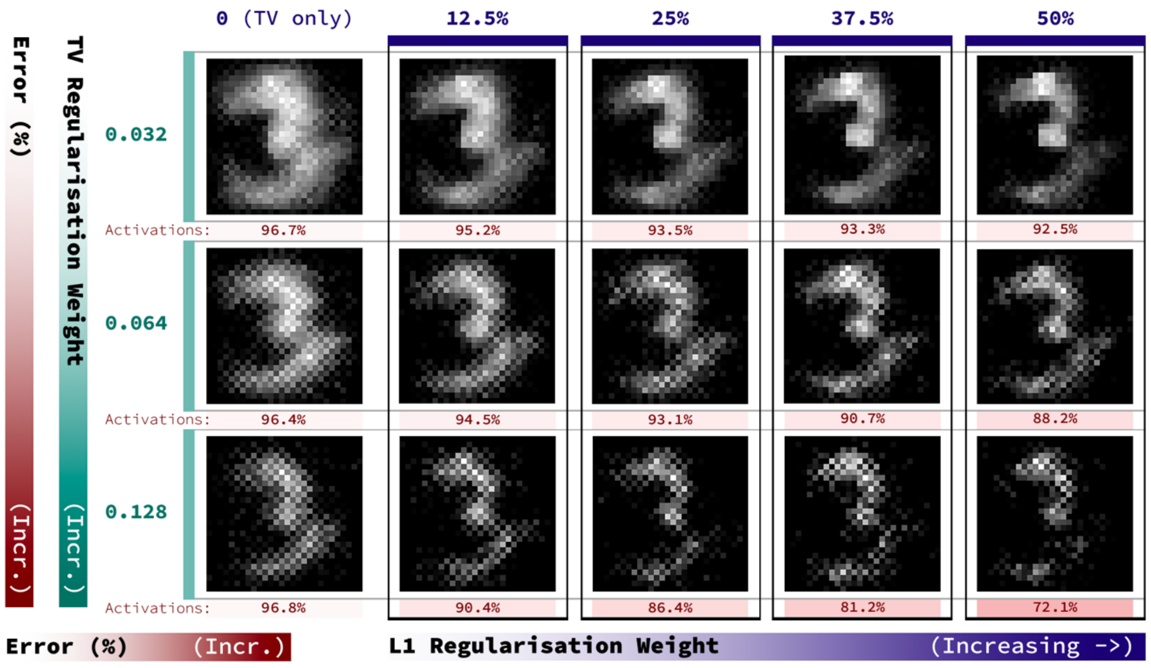

Figure 4. Images optimised for activation of the output neuron 3, with varying levels of TV and L1 regularisation. The former increases downwards (exponentially) and the latter increases rightwards (linearly). The L1 regularisation weights are written as a percentage of the TV regularisation weight for each row. All images were initialised from near-zero values and optimised for 100 steps. The activation induced by each image is written below it in red, in a red box with opacity equal to the error, i.e., 1 minus the activation.

Following this, one final regularisation technique was added in an attempt to get even closer to actual MNIST images. Specifically, this involved jittering the image before evaluating the loss and gradients, i.e., translating it 1 pixel along either or both axes, to select for images were robust to such small transformations. Unfortunately, jittering regularisation by itself did not result in interpretable images and combining it with other techniques such as TV regularisation proved too much for the optimiser, which then invariably failed (for output or hidden nodes). As such, jittering will not be addressed in detail here. It is, however, worth mentioning the behaviour of the optimiser in this case, which typically found only small maxima (unsatisfactory on all measures) and would often diverge towards images which maximised some of the measures but performed poorly on others. This suggests that images meeting all constraints, i.e., smooth images which were robust to small translations and uniquely activated the target node, were either few and far between, or didn't exist at all.

Despite the failure of adding more regularisation techniques, optimisation with L1 and/or TV regularisation seemed to be sufficiently reliable for output neurons to warrant utilising it on the hidden neurons as well. This was initially done without penalising activation of other nodes, and that method was ultimately maintained for reasons explained in more detail below.

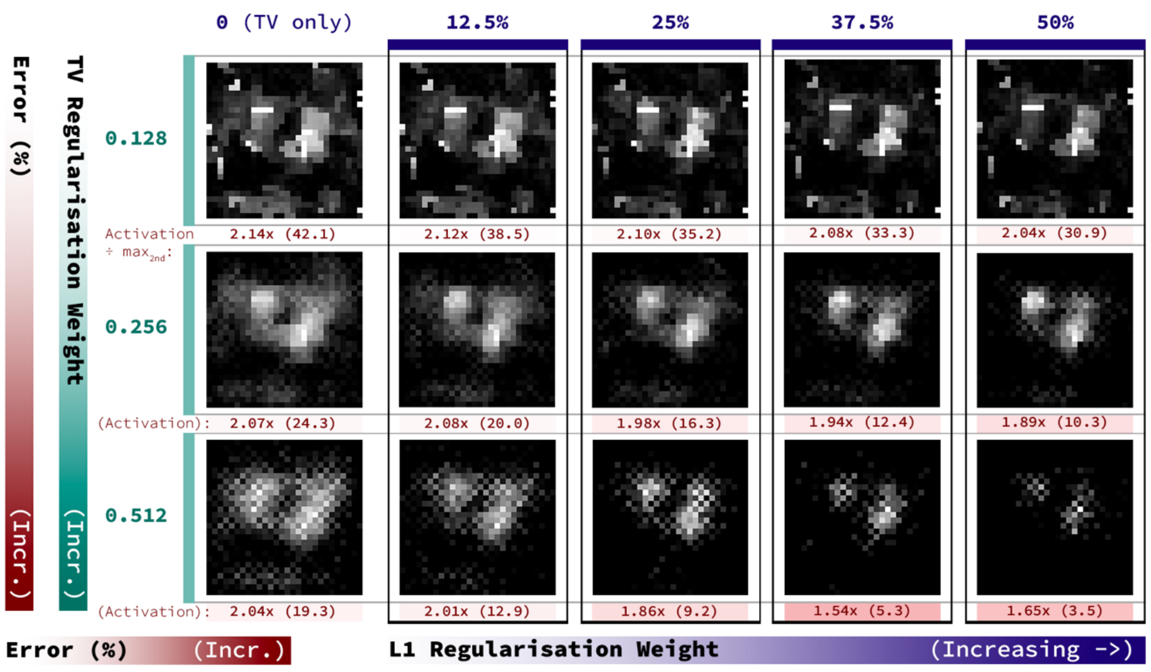

Figure 5. Images optimised for activation of neuron 1 in hidden layer 7, with TV regularisation increasing downwards (exponentially) and L1 regularisation increasing rightwards (linearly). L1 weights are written as percentages of TV weights again. All images were initialised from near-zero values and optimised for only 30 steps, to limit extreme values. Written below each image is its activation of the target node divided by the next-highest node activation, with the raw target node activation in brackets. The opacity of the red boxes underneath the activation values is 1 minus that images' activation ratio divided by the maximum ratio 2.14. These opacities are very similar to those in Figure 4, indicating that the regularisations had a similar effect on error.

Optimising for hidden neurons required some fine-tuning of hyper-parameters, especially the TV and L1 weights, and the number of steps. The amounts of these parameters that were most favourable for interpretability seemed to differ between each layer, particularly between the hidden and output layers. Additionally, optimisation typically progressed more quickly for later (downstream) hidden layers with less nodes, and was more conducive to stronger regularisation, while optimisation was more liable to fail in earlier (upstream) layers, with the same regularisation weightings. In the first few hidden layers it was sometimes even necessary to remove one regularisation technique entirely, to achieve high activation of the target node. This was likely partially because activations are generally smaller in earlier layers and increase gradually as more and more neuron outputs are summed together. But it also appeared that the neurons in later layers were responding to more concentrated features which more naturally conformed to sparse and smooth representations, while the neurons in earlier layers were responding to more widely distributed regions or pixels all around the 28x28 space. Hence, the images that activated neurons on the latter end of that spectrum were naturally denser and noisier. This also meant that it was harder to target individual neurons or small groups of neurons in the earlier layers (compared to later layers), since more neurons would tend to be activated by the same pixels.

It was ascertained at this point that penalising the activation of other nodes in the layer had only an insignificant effect on the optimised images, so it was kept at 0 for all visualisations henceforth. Evidently the combination of TV and L1 regularisation had sufficiently narrowed the space of possible images such that the optimiser converged on consistent patterns regardless of penalisation and even regardless of initialisation. The images with these patterns tended to activate multiple nodes. So, a harsh penalty term for other activations would eventually start to push down the target node's activation as well, and even a mild penalty would simply limit the growth rate of all activations without significantly altering the ratios. The optimiser was essentially unable to generate interpretable images that activated *only* one node in a given hidden layer. This posed a contrast to the output layer where unique activation happened automatically, but only because of the SoftMax activation function. Indeed, the activation of some nodes in the hidden layers seemed to consistently correlate together in clusters, such that if any of them were optimised for, all the nodes in the cluster would activate somewhat. This suggests that the neurons were working synergistically – perhaps to integrate features together, e.g., components of a particular digit.

This was not inherently a problem, though, since the images would typically still activate the target node significantly more than any other node. Hence the patterns, which were unique to each neuron, could offer insight about their functions. Indeed, many of the images generated, especially when targeting later hidden layers, were as interpretable as the output neuron visualisations (given no prior knowledge of the output neuron functions) and could reasonably be expected to be as informative as those visualisations. Most of the information pertained to what regions of the 28x28 space were important for activation of each node. Some examples are shown in Figures 6 and 7. It is impossible to address or even summarise accurately the visualisations of all the thousands of neurons here. It would be overly lengthy even to go through all 32 neurons in the final hidden layer – so the reader is encouraged to use the appended code to visualise more hidden neurons at their discretion.

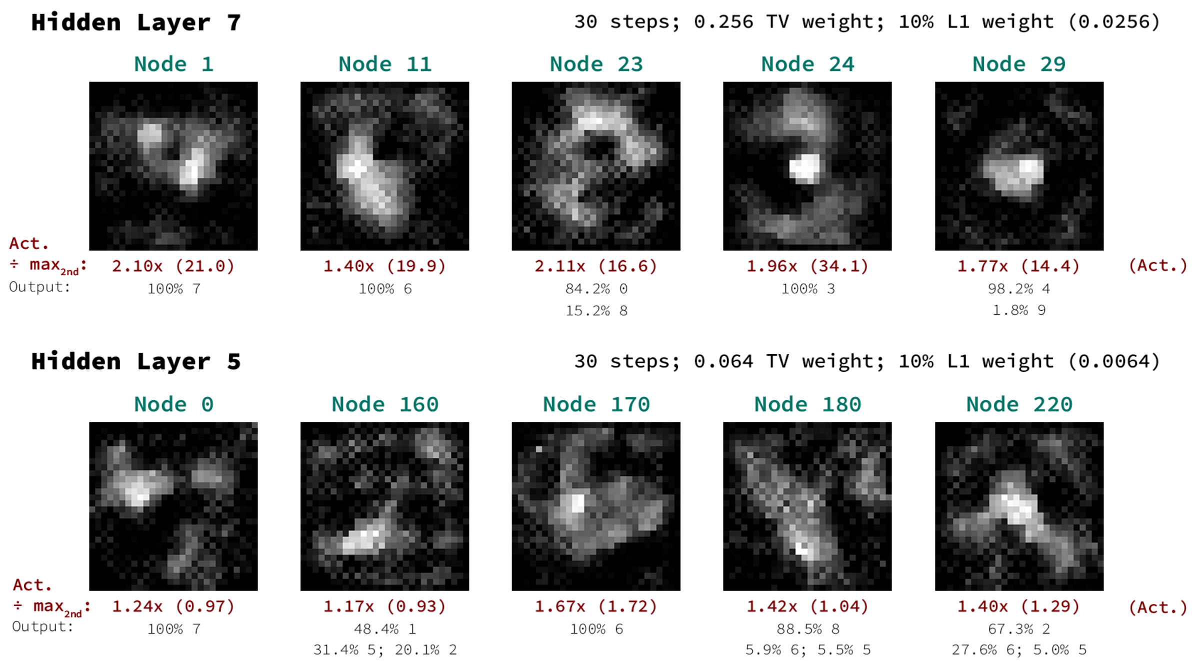

Figure 6. Visualisations of select nodes in hidden layers 7 and 5, with some different hyper-parameters, specified above each set of images. Below each image is written (in red) the ratio of its activation of the target to the next-highest node activation, with the raw target node activation in brackets. Further below these in light black font are the top 3 activations induced in the output layer when querying the DNN again with the visualisation.

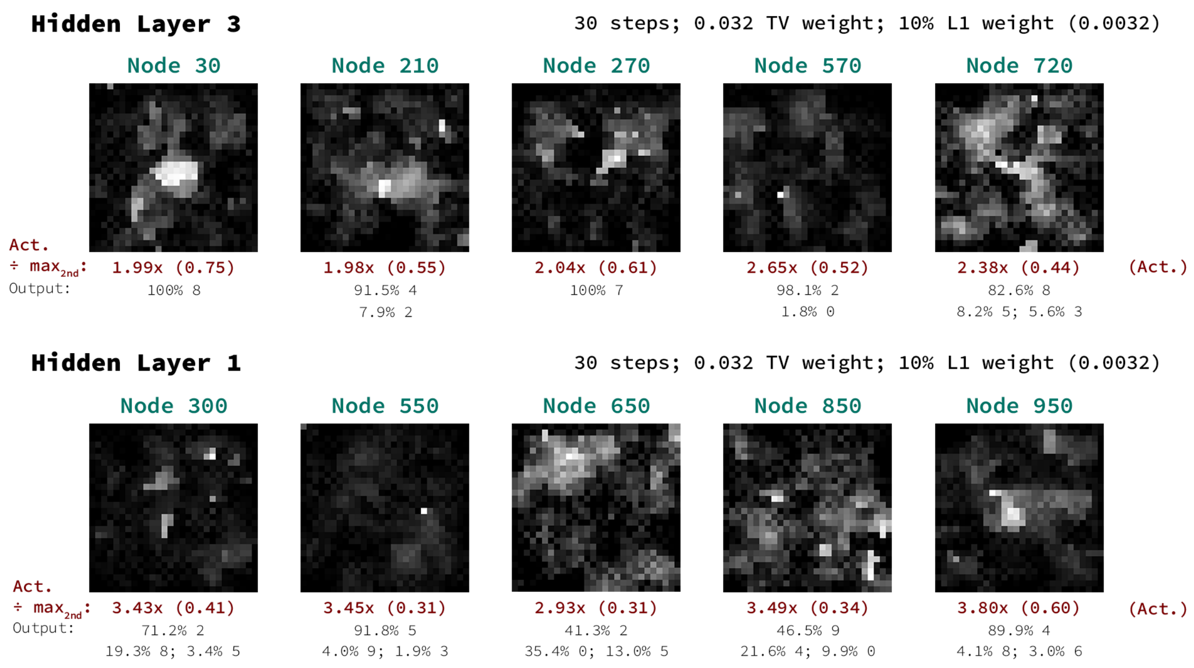

Figure 7. Visualisations of select nodes in hidden layers 3 and 1. All quantities are displayed in the same format as Figure 6.

The network was also queried with the visualisations to ascertain its induced activations of output nodes, and therefore, which output nodes correlated most strongly to the features relevant to that hidden neuron. It was found that neurons in earlier hidden layers had more distributed output activations, whereas for later layers there were more concentrated. For instance, visualisations from the last hidden layer tended to activate only one output neuron very strongly. This corroborated the previous evidence in support of neurons gradually becoming more specialised across the hidden layers.

It bears mentioning also that some neurons (a minority) seemed to be ‘dead’ – that is, their activation could not be increased beyond 0, even with random initialisation and many optimisation steps, and they would not respond to any of the images in the MNIST data either. This is common with the ReLU activation function since a neuron that always receives negative inputs will always output 0. It indicates that the DNN had learnt during training not to utilise those neurons. However, it cannot be deduced from this whether performance would be higher or lower with a different architecture which forces the DNN to utilise all neurons (e.g., a different activation function or a variant of ReLU).

This information does not necessarily form a comprehensive account of each neuron – it does not reveal, for instance, exactly which MNIST images have the features that induce high activation of the neuron. Indeed, that could potentially be ascertained by simply querying the hidden layers with the MNIST images. But the optimisation method says far more about *why* each image activates the neurons that it does, and therefore, more about *how* the DNN’s hidden layers are processing the images – both aspects of classification that were more at the core of the research question.

Further Discussion:

Returning to the initial motivations for this project, we can conclude that the hypothesis, as stated, was incorrect. Evidently, the DNN did not seem to develop neurons with the capacity to identify very generalised features, such as all the instances of horizontal lines in the MNIST images. This may be due to strong differences between the MNIST training data, and the training data used for larger, convolutional image classifiers, which would typically include a much wider range of images, including variants of the same types of images which differ by transformations (e.g., translations or rotations). This forces the network to learn to recognise the underlying geometric features that are common to many images. Hence, optimising for its neurons can result in, for instance, a distinct geometric pattern which tiles the image space (Olah, Mordvintsev, & Schubert, 2017). In contrast to this, all the features in the MNIST data set are found in specific spatial locations, and indeed, the locations are more important (for MNIST) than the exact geometric nature of the feature, so that any images with extreme pixels in that location can activate the relevant neuron, even if they're very noisy and unrecognisable.

This is perhaps counterintuitive since it would be more difficult for a human to cognitively process and identify a digit by looking individually at smaller regions of the image, rather than simply looking at the whole image at once. And this would probably be unusual for a more powerful DNN as well. But for the MNIST test set, it seems that even simple heuristics, like this gradual integration of parts, are sufficient to classify over 98% of images correctly. Given that, it is no wonder if this is what the hidden neurons do.

Another way to view this is that the ontology represented by the MNIST classifier is very minimal, since it depends entirely upon the MNIST training data. It is blind to any categories that don't exist within the training data, e.g., images that depict objects other than digits, or even images that don't depict anything at all. So, these are processed the exact same way as MNIST images, and may, for instance, be classified as digits. The classifier is like a hammer that treats every input like a nail – even though the vast majority of possible 28x28 monochromatic images are nothing like MNIST images. Hence the image features that the classifier learns are important are not necessarily important outside of the context of MNIST, i.e., they do not generalise effectively.

This is also why regularisation techniques must be used to filter for images which demonstrate neuron activation in a more visually meaningful way – 'visually meaningful' is a category imposed by human brains, not a category that is inherent in the mathematical structure of 28x28 arrays of pixels. But this is the ultimate task of both the human analysis of neural networks and the neural networks themselves: to make sense of all the data, and to bring out the signal from amidst all the noise. The brilliance of neurons, whether real or artificial, is in how they come together to learn anything at all.

References

Caselles, V., Chambolle, A., & Novaga, M. (2015). Total Variation in Imaging. In Scherzer, O. (Eds.), *Handbook of Mathematical Methods in Imaging* (pp. 1455–1499). Springer. https://doi.org/10.1007/978-1-4939-0790-8_23

Ciresan, D. C., Meier, U., Gambardella, L. M., & Schmidhuber, J. (2010). Deep, Big, Simple Neural Nets for Handwritten Digit Recognition. *Neural Computation*, *22*(12), 3207–3220. http://doi.org/10.1162/NECO_a_00052

Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep Sparse Rectifier Neural Networks. In G. Gordon, D. Dunson, & M. Dudík (Eds.), *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (pp. 315–323). PMLR. https://proceedings.mlr.press/v15/glorot11a.html

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *Proceedings of the IEEE/CVF International Conference on Computer Vision* (pp. 1026–1034). IEEE. https://doi.org/10.48550/arXiv.1502.01852

Mahendran, A., & Vedaldi, A. (2015). Understanding Deep Image Representations by Inverting Them. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 5188–5196). IEEE. https://doi.org/10.48550/arXiv.1412.0035

Olah, C., Mordvintsev, A., & Schubert, L. (2017). Feature Visualisation. *Distill*. https://doi.org/10.23915/distill.00007

Rudin, L. I., Osher, S., & Fatemi, E. (1992). Nonlinear total variation based noise removal algorithms. *Physica D: Nonlinear Phenomena, 60*(1-4), 259–268. https://doi.org/10.1016/0167-2789(92)90242-f

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological), 58*(1), 267-288. https://doi.org/10.1111/j.2517-6161.1996.tb02080.x

Appendix: Formulas

1. Formulas of activation functions, loss function, and their derivatives in the classifier

For any real number x:

$$\mathrm{ReLU}(x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases} \text{ so } \mathrm{ReLU}'(x) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases}$$

Note: the following formulas assume batch size 1 for simplicity.

$$\text{Let } \vec{z} = \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix}, \text{then } \mathrm{SoftMax}(\vec{z}) = \begin{bmatrix} \mathrm{SoftMax}(z_1) \\ \vdots \\ \mathrm{SoftMax}(z_n) \end{bmatrix}$$

$$\text{And for } 1 \leq k \leq n, \mathrm{SoftMax}(z_k) = \frac{\exp(z_k - \max\{z_1, \cdots, z_n\})}{\sum_{i=1}^{n} \exp(z_i - \max\{z_1, \cdots, z_n\})}$$

Also note max terms are only for numerical stability in code.

$$\Rightarrow \mathrm{SoftMax}(z_k) = \frac{\exp(-\max\{z_1, \cdots, z_n\}) * \exp(z_k)}{\exp(-max\{z_1, \cdots, z_n\}) * \sum_{i=1}^{n} \exp(z_i)} = \frac{\exp(z_k)}{\sum_{i=1}^{n} \exp(z_i)}$$

Let the cross– entropy of an output vector $\vec{y}$ with respect to a target vector $\vec{t}$ be $CE(\vec{y}, \vec{t})$.

$$\Rightarrow CE(\vec{y}, \vec{t}) = -\sum_{i=0}^{n} t_i \log(y_i) = -\log(y_t) = CE(y_t, \vec{t})$$

where y_t is the activation of the target node.

$$\Rightarrow CE(z_t, \vec{t}) = -\log(SoftMax(z_t)) = -\log\left(\frac{\exp(z_t)}{\sum_{i=1}^{n} \exp(z_i)}\right)$$

$$\Rightarrow CE(z_t, \vec{t}) = -\log(\exp(z_t)) + \log\left(\sum_{i=1}^{n} exp(z_i)\right) = \log\left(\sum_{i=1}^{n} exp(z_i)\right) - z_t$$

where z_t is the input to the target node.

So, for all $k \neq t$:

$$\frac{dCE}{dz_k} = \frac{1}{\sum_{i=1}^{n} exp(z_i)} * \exp(z_k) = \mathrm{SoftMax}(z_k)$$

And (for $k = t$):

$$\frac{dCE}{dz_t} = \frac{1}{\sum_{i=1}^{n} exp(z_i)} * \exp(z_t) - 1 = \mathrm{SoftMax}(z_t) - 1$$
$$= -\frac{1}{n} * \frac{1}{y_t} * \mathrm{SoftMax}'(z_k) = -\frac{1}{n} * \frac{1}{y_t} * \mathrm{SoftMax}'(z_k)$$

So:

$$\frac{\partial CE}{\partial \vec{z}} = \begin{bmatrix} \frac{dCE}{dz_1} \\ \vdots \\ \frac{dCE}{dz_t} \\ \vdots \\ \frac{dCE}{dz_n} \end{bmatrix} = \begin{bmatrix} \mathrm{SoftMax}(z_1) \\ \vdots \\ \mathrm{SoftMax}(z_t) - 1 \\ \vdots \\ \mathrm{SoftMax}(z_n) \end{bmatrix} = \begin{bmatrix} \mathrm{SoftMax}(z_1) \\ \vdots \\ \mathrm{SoftMax}(z_t) \\ \vdots \\ \mathrm{SoftMax}(z_n) \end{bmatrix} - \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} = \mathrm{SoftMax}(\vec{z}) - \vec{t}$$

$$\text{I.e. } \frac{\partial CE}{\partial \vec{z}} = \vec{y} - \vec{t} \Rightarrow -\frac{\partial CE}{\partial \vec{z}} = \vec{t} - \vec{y}$$

Derivative of SoftMax w.r.t. layer inputs for visualisation:

For all $k \neq t$ (by quotient rule):

$$\frac{\mathrm{dSoftMax}(z_k)}{dz_t} = \frac{\frac{d\exp(z_k)}{dz_t} * \sum_{i=1}^{n}\exp(z_i) - \exp(z_k) * \frac{d\sum_{i=1}^{n}\exp(z_i)}{dz_t}}{\left(\sum_{i=1}^{n}\exp(z_i)\right)^2}$$

Hence:

$$\frac{\mathrm{dSoftMax}(z_k)}{dz_t} = \frac{0 * \sum_{i=1}^{n}\exp(z_i)}{\left(\sum_{i=1}^{n}\exp(z_i)\right)^2} - \frac{\exp(z_k) * \exp(z_t)}{\sum_{i=1}^{n}\exp(z_i) * \sum_{i=1}^{n}\exp(z_i)}$$

So:

$$\frac{\mathrm{dSoftMax}(z_k)}{dz_t} = -\mathrm{SoftMax}(z_k) * \mathrm{SoftMax}(z_t)$$

And (for $k = t$):

$$\frac{\mathrm{dSoftMax}(z_t)}{dz_t} = \frac{\frac{d\exp(z_t)}{dz_t} * \sum_{i=1}^{n}\exp(z_i) - \exp(z_t) * \frac{d\sum_{i=1}^{n}\exp(z_i)}{dz_t}}{\left(\sum_{i=1}^{n}\exp(z_i)\right)^2}$$

Hence:

$$\frac{\mathrm{dSoftMax}(z_t)}{dz_t} = \frac{\exp(z_t) * \sum_{i=1}^{n}\exp(z_i) - \exp(z_t) * \exp(z_t)}{\left(\sum_{i=1}^{n}\exp(z_i)\right)^2} = \frac{\exp(z_t)}{\sum_{i=1}^{n}\exp(z_i)} - \frac{(\exp(z_t))^2}{\left(\sum_{i=1}^{n}\exp(z_i)\right)^2}$$

So:

$$\frac{\mathrm{dSoftMax}(z_t)}{dz_t} = \mathrm{SoftMax}(z_t) * \left(1 - \mathrm{SoftMax}(z_t)\right)$$

Therefore:

$$\frac{\partial \mathrm{SoftMax}(\vec{z})}{\partial z_t} = \begin{bmatrix} \frac{\mathrm{dSoftMax}(z_1)}{dz_t} \\ \vdots \\ \frac{\mathrm{dSoftMax}(z_t)}{dz_t} \\ \vdots \\ \frac{\mathrm{dSoftMax}(z_n)}{dz_t} \end{bmatrix} = \begin{bmatrix} -\mathrm{SoftMax}(z_1) * \mathrm{SoftMax}(z_t) \\ \vdots \\ \mathrm{SoftMax}(z_t) * \left(1 - \mathrm{SoftMax}(z_t)\right) \\ \vdots \\ -\mathrm{SoftMax}(z_n) * \mathrm{SoftMax}(z_t) \end{bmatrix}$$

And:

$$J_{\mathrm{SoftMax}} = \frac{\partial \mathrm{SoftMax}(\vec{z})}{\partial \vec{z}} = \begin{bmatrix} \frac{\partial \mathrm{SoftMax}(\vec{z})}{\partial z_1} & \cdots & \frac{\partial \mathrm{SoftMax}(\vec{z})}{\partial z_t} & \cdots & \frac{\partial \mathrm{SoftMax}(\vec{z})}{\partial z_n} \end{bmatrix}$$

This Jacobian matrix is used in the code to backpropagate activation of the output nodes.

2. Partial derivative of a target layer k and a target neuron t in that layer, with respect to the inputs to the DNN

Let I be the input image.
Let z_m^n be the weighted input for the m^{th} neuron in the n^{th} layer, and z^n be the full inputs.
Let $f(x)$ be the activation function for a given layer (e.g. ReLU), so $f'(x)$ is the derivative.
(NB: the function is usually the same, but could differ, e.g., for the output layer.)
Then $A = f(z^k)$ is the target layer activation, and $A_t = f(z_t^k)$ is the target neuron activation.
Hence,

$$\frac{\partial A}{\partial I} = \frac{\partial f(z^k)}{\partial I} = \frac{\partial f(z^k)}{\partial z^k} * \frac{\partial z^k}{\partial f(z^{k-1})} * \frac{\partial f(z^{k-1})}{\partial z^{k-1}} * \frac{\partial z^{k-1}}{\partial f(z^{k-2})} * \cdots * \frac{\partial f(z^1)}{\partial z^1} * \frac{\partial z^1}{\partial I}$$
$$\Rightarrow \frac{\partial A}{\partial I} = f'(z^k) * w^k * f'(z^{k-1}) * w^{k-1} * \cdots * f'(z^1) * w^1$$

and

$$\frac{\partial A_t}{\partial I} = \frac{\partial f(z_t^k)}{\partial I} = \frac{\partial f(z_t^k)}{\partial z_t^k} * \frac{\partial z_t^k}{\partial f(z^{k-1})} * \frac{\partial f(z^{k-1})}{\partial z^{k-1}} * \frac{\partial z^{k-1}}{\partial f(z^{k-2})} * \cdots * \frac{\partial f(z^1)}{\partial z^1} * \frac{\partial z^1}{\partial I}$$
$$\Rightarrow \frac{\partial A}{\partial I} = f'(z_t^k) * w_{:,t}^k * f'(z^{k-1}) * w^{k-1} * \cdots * f'(z^1) * w^1$$

where $w_{:,m}^n$ is the mth column only of the weights to the nth layer, and w^n is the full weights.

3. numpy.clip formulas

$$\mathrm{f}(x) = \begin{cases} 0, & x \le 0 \\ x, & 0 < x < 1 \\ 1, & x \ge 1 \end{cases}, \text{so } f'(x) = \begin{cases} 0, & x \le 0 \\ 1, & 0 < x < 1 \\ 0, & x \ge 1 \end{cases}$$

4. Derivation of TV formula derivative for a pixel $u_{a,b}$

$$TV(u) = \sum_{\substack{1 \le i \le N \\ 1 \le j \le N}} |u_{i+1,j} - u_{i,j}| + \sum_{\substack{1 \le i \le N \\ 1 \le j \le N}} |u_{i,j+1} - u_{i,j}|$$

$$= (|u_{2,1} - u_{1,1}| + \cdots + |u_{2,b} - u_{1,b}| + \cdots + |u_{2,N} - u_{1,N}| + \cdots + |u_{2,b} - u_{1,b}| + \cdots$$
$$+ |u_{a,b} - u_{a-1,b}| + |u_{a+1,b} - u_{a,b}| + \cdots + |u_{N,b} - u_{N,b}|)$$
$$+(|u_{1,2} - u_{1,1}| + \cdots + |u_{a,2} - u_{a,1}| + \cdots + |u_{N,2} - u_{N,1}| + \cdots + |u_{a,2} - u_{a,1}| + \cdots$$
$$+ |u_{a,b} - u_{a,b-1}| + |u_{a,b+1} - u_{a,b}| + \cdots + |u_{a,N} - u_{a,N}|)$$

Hence:

$$\frac{dTV}{du_{a,b}} = \frac{d}{du_{a,b}}(|u_{a,b} - u_{a-1,b}| + |u_{a+1,b} - u_{a,b}|)$$
$$+ \frac{d}{du_{a,b}}(|u_{a,b} - u_{a,b-1}| + |u_{a,b+1} - u_{a,b}|)$$

Therefore:

$$\frac{dTV}{du_{a,b}} = \mathrm{sgn}(u_{a,b} - u_{a-1,b}) - \mathrm{sgn}(u_{a+1,b} - u_{a,b})$$
$$+\mathrm{sgn}(u_{a,b} - u_{a,b-1}) - \mathrm{sgn}(u_{a,b+1} - u_{a,b})$$

5. Derivation of TV derivative code implementation

Firstly, a padded version of the image is created using numpy.pad, which simply adds a blank margin of one pixel along the entire image, i.e., it adds 1 row or column of 0s to each side of the image, thereby creating a 30x30 image. This will be hereby referred to as `img_pad`, and the original image will be referred to as `img`.

Indexing `img_pad` to remove 2 rows or columns from any side (and keeping the original 28 rows / columns along the perpendicular axis) is equivalent to translating the original image 1 pixel towards that side. E.g., `img_pad[1:-1, :-2]` is equivalent to translating the original image 1 pixel towards the right side, hence each pixel in the image $u'_{a,b}$ is equal to $u_{a,b-1}$, where u is the original image.

Therefore:

$$\frac{dTV}{du_{a,b}}$$
$$= \operatorname{sgn}(u_{a,b} - u_{a-1,b}) - \operatorname{sgn}(u_{a+1,b} - u_{a,b}) + \operatorname{sgn}(u_{a,b} - u_{a,b-1}) - \operatorname{sgn}(u_{a,b+1} - u_{a,b})$$

$$= \operatorname{sgn}\left(u_{a,b} - \texttt{img\_pad[:-2, 1:-1]}_{a,b}\right) - \operatorname{sgn}\left(\texttt{img\_pad[2:, 1:-1]}_{a,b} - u_{a,b}\right)$$
$$+\operatorname{sgn}\left(u_{a,b} - \texttt{img\_pad[1:-1, :-2]}_{a,b}\right) - \operatorname{sgn}\left(\texttt{img\_pad[1:-1, 2:]}_{a,b} - u_{a,b}\right)$$

This formula is the same for all pixels $u_{a,b}$. Hence:

$$\frac{dTV}{du} = \operatorname{sgn}(u - \texttt{img\_pad[:-2, 1:-1]}) - \operatorname{sgn}(\texttt{img\_pad[2:, 1:-1]} - u)$$
$$+\operatorname{sgn}(u - \texttt{img\_pad[1:-1, :-2]}) - \operatorname{sgn}(\texttt{img\_pad[1:-1, 2:]} - u)$$

$$= \operatorname{sgn}(u - \texttt{img\_pad[:-2, 1:-1]}) + \operatorname{sgn}(u - \texttt{img\_pad[2:, 1:-1]})$$
$$+\operatorname{sgn}(u - \texttt{img\_pad[1:-1, :-2]}) + \operatorname{sgn}(u - \texttt{img\_pad[1:-1, 2:]})$$

$$= \operatorname{sgn}(\texttt{img} - \texttt{img\_pad[:-2, 1:-1]}) + \operatorname{sgn}(\texttt{img} - \texttt{img\_pad[2:, 1:-1]})$$
$$+\operatorname{sgn}(\texttt{img} - \texttt{img\_pad[1:-1, :-2]}) + \operatorname{sgn}(\texttt{img} - \texttt{img\_pad[1:-1, 2:]})$$

This final formulation is used in the code.