Intro to Deep Learning Assignment 1

Student Number:13064947, email:john.goodacre.13@ucl.ac.uk

November 2017

1 Deep Learning and Features: Question 1

What does it mean when people say deep learning does not rely on explicit features?

Much of machine learning, particularly supervised learning outside the world of Deep Learning relies upon feature engineering. In other words for whatever task the data scientist spends a great deal of time and effort calculating, guessing or dreaming features that may have a predictive influence upon the data.

The world of machine vision is an interesting case in point. A lot of machine vision expertise previously comprised working with things like edge detectors, finding interest points and descriptors around those points. Also the images would often be pre-processed, for example histogram equalisation. The pre-processed images together with their features would then be put into a statistical model for whatever regression or classification task was required.

Deep Learning takes a different approach. Sticking with the vision example, the Deep Learning approach is closer to an end to end system in that raw pixels may be put in as training data, with a pre-defined architecture. Then the model is trained from scratch.

The reality is a little more complex in that although for example fully connected feed forward networks have good properties for many vision tasks, a more restricted version - The convolutional neural network has had state of the art results. In some ways, the features have become embedded in the architecture.

For example, in traditional machine vision, edge detectors such as Sobel used kernels. In a Convolutional Neural Network many kernels are used in an automatic fashion creating multiple feature maps. Typically edge detectors come out as an important 'feature' in the first layer. By then using pooling the neural network can learn where these structures have the highest activations. By sharing weights the neural network can learn these structures regardless of location in the image. In contrast if in some perverse world only pixels very far apart had

a relationship to one another then using small window kernels in convolutional neural nets would presumably make them far less capable!

Instead learns a hierarchy of representations (i.e., features) from the data?

Vision again is a very good example here. Visual data by its nature appears to naturally suit the function composition implicit in Deep Learning. A deep neural network is effectively a composition of functions, non-linear functions. It appears that high dimensional data such as visual data, may lie close to a much lower dimensional manifold. Deep Learning applies repeated non-linear function composition and thus has very expressive power in such situations.

One way I enjoy thinking about this is through Auto-Encoders. In a linear form we have pure PCA, but with non-linear activations we have much more. Anyway the point is that whenever a deeper layer has fewer neurons than a prior layer we are effectively learning a lower dimensional representation of that data. High dimensional, highly structured, low(ish) noise data such as vision should thus be able to be compressed, each successive representation can learn repeated structure embedded within the data itself.

Describe one benefit of this approach?

My own subjective opinion is that if we pre-impose our own view on the data and do feature engineering, we can in turn by manipulating the data within the realms of our pre-conceptions end up losing some viable information. In contrast to letting the data speak for itself. A second more practical one is that one doesn't need an expensive domain expert to tell us what features we should be constructing.

Give an example of a particular feature that was previously popular (perhaps from vision or speech) but now is less frequently used due to the adoption of deep architectures. Describe in one sentence the key idea behind the design of this feature and ability to help in class discrimination.

I gave the example of edge detectors and points of interest (together with their descriptors). Here is my one sentence. Edge detectors now come out naturally from the first layer of a Convolutional neural network where we choose our kernel size and number of feature maps - Sobel edge detectors amongst other older approaches where we pre-impose features that we expect to help with vision classification tasks, have been subsumed by Deep Learning. (I did cheat I guess, there should be a full stop between feature maps and Sobel. But one sentence is tough).

2 Look at F1/ train/ test time on variations of choice: Question 2

Report in a table what differences you observed in terms of the F1-score between the models under three variations of your choice.

Finding a best classifier with only a few experiments is probably a bit too much to illustrate with three variations. In practice, I would tend to think through my architecture first and later on do the usual train/ validate/ test across some hyper-parameter search in order to attempt to approach a 'best' predictor.

Given we are comparing several different models, with different architectures, rather than attempt with only three examples to find a 'best' set of hyper-parameters as regards loss and ideally classification accuracy, I was instead more curious about using each given model as a baseline and then changing the size of the model significantly - the aim being to take a peek at predictive ability against train/test time. I also noticed that dropout was set to 0.5 and wished to observe the impact of dropout on generalisation ability. The exception was CNN's where I played around with feature maps somewhat.

2.1 Experimental setup

This is slightly artificial as we are comparing hyper-parameters across differing models with different parameters. But I will lay out the common parameters I left unchanged for the optimisations. The structure of the model was typically altered by making it deeper or wider. Clearly the optimisation parameters, including batch sizes can have a significant effect of training time however I focused more on architecture.

- Training Epochs: 3
- Optimizer: Adam
- Loss Metric: Categorical Cross Entropy
- Activation function: Relu
- Output layer: Softmax
- Batch Size: 128
- Data:Window 100, Train/ Val split: 9466/3478

2.2 Multi-Layer Perceptron

The baseline MLP is the one given to us. It has 683,788 trainable parameters and gave an F1 on the validation set of 0.72. The experiments performed were to multiply the number of hidden layer neurons by 4. To add (and remove) a layer of depth and dropout and to alter the dropout percentage to 0.25 and 0.0.

Bar removing dropout (which was a bad idea for generalisation), results were reasonably stable as regards F1 on the validation set. Basically a wider or deeper neural net added nothing, but making the net shallower or narrower reduced the F1. The key differences were seen in training and inference times. Clearly with 512 hidden neurons the network took nearly 4x as long to train. The strongest result as regards dropout was that the training F1 went up significantly but validation F1 reduced when dropout was removed. Basically the net 'learned' the training examples but had poorer generalisation ability.

MLP	Hyperparameters	Training Time	Inference Time	Mean F1
Baseline	683,788	2.99	165 millisecs	0.72
WideMLP	2,931,724	11.2	438 millisecs	0.71
NarrowMLP	167,884	1.86	146 millisecs	0.67
DeeperMLP	700,300	3.43	236 millisecs	0.73
ShallowMLP	667,276	2.87	171 millisecs	0.66
MLP(D/O 0.25)	683,788	3.15	176 millisecs	0.71
MLP(D/O 0.00)	683.788	2.74	161 millisecs	0.67

Table 1: MLP

2.3 Long Short Term Memory

The structural experiments as regards the LSTM were to stack another LSTM, remove an LSTM and change the number of hidden neurons to 32 and 128 respectively. Again I played with dropout. As regards results, clearly the LSTM takes far longer to train due to its sequential nature. Inference is again slower. There were no real surprises - removing dropout was bad for generalisation, and wider or deeper LSTMs had similar proportional effects as regards train/ test time to the previous example. If I had guessed in advance I would have expected the F1 scores to be somewhat better than the MLP. However on this architecture they were worse.

2.4 Convolutional Neural Network

The CNN had excellent F1 scores on the validation set. Instead of drastically changing the architecture of the CNN the experiments here focused on altering the number of feature maps, changing the kernel size and altering the max

LSTM	Hyperparameters	Training Time	Inference Time	Mean F1
Baseline	63,756	2min 21 secs	9.69 secs	0.67
LSTM128	225,804	3min 49 secs	11.4 secs	0.66
LSTM32v	19,596	1min 44 secs	6.99 secs	0.62
VanillaLSTM	30,732	1min 6 secs	4.62 secs	0.65
3StackLSTM	96,780	3min 26 secs	12 secs	0.64
LSTM64(0.25)	63,756	2min 18 secs	8.42 secs	0.67
LSTM64(0.0)	63,756	2min 20 secs	8.46 secs	0.62

Table 2: LSTM

pooling size. (32 and 8 feature maps, a 1x10 kernel and 1x3 max pool size). Again I fiddled with the dropout proportion. Because of the heavier computational load, clearly the CNN takes longer to train and infer than the multi-layer perceptron, albeit with fewer hyper-parameters and thus a smaller memory load.

In terms of compute the CNN lies somewhere between the MLP and LSTM. Basically the most promising result was that halving the number of feature maps nearly halved the training and inference time with no adverse effect on an already high F1.

CNN	Hyperparameters	Training Time	Inference Time	Mean F1
BaselineCNN	667,420	1min 10 secs	1.89 secs	0.84
CNNF32	1,336,972	2min 13 secs	4.21 secs	0.84
CNNF8	333,604	39.6	1.15 secs	0.84
CNNK(1,10)	668,780	1min 28secs	2.76 secs	0.81
CNNM(1,3)	294,684	50.3 secs	1.41 secs	0.79
CNN(0.25)	667,420	1min 10 secs	1.92 secs	0.86
CNN(0.0)	667,420	1min	1.92 secs	0.81

Table 3: CNN

2.5 Conv LSTM

The hybrid Convolutional neural net/LSTM instead replaces the fully connected layer with an LSTM. I experimented with adding another LSTM and changing the number of feature maps. Here I also started changing the batch sizes, mainly to see the effect on training time. Again I played with dropout. Quick summary - reducing batch sizes increases training time for each epoch as one would expect. Stacking the LSTMs did not improve generalisation ability. Changing the number of feature maps did little to predictive ability bar increase compute time and removing dropout was as usual a pretty bad idea.

Discuss the reasons for the observed differences with respect to

CNNLSTM	Hyperparameters	Training Time	Inference Time	Mean F1
BaseCNNLSTM	112,508	1min 42 secs	3.44 secs	0.88
CNNLSTMx2	120,828	1min 59 secs	4.53 secs	0.79
CNNLSTMB64	112,508	1min 53 secs	3.51 secs	0.90
CNNLSTMB32	112,508	2min 22 secs	3.47 secs	0.87
CNNLSTMF32	222,956	3min 3 secs	6.15 secs	0.88
CNNLSTM(0.25)	112,508	1min 41 secs	3.59 secs	0.87
CNNLSTM(0.0)	112,508	1min 35 secs	3.85 secs	0.85

Table 4: CNN LSTM

accuracy and efficiency (e.g., training time, inference time, number of parameters).

Much has been discussed above, but quick summary. For generalisation (F1 on the validation set), choosing the right architecture is the main thing - CNN's were good for this task. There is a trade off between memory load and compute between CNNs and MLPs - i.e fully connected layers have (m x n) number of connections where m is the number of neurons in layer (l-1) and n the number of layers in layer l. Whereas CNNs have the compute load of calculating kernels across all segments of data, across all channels, across the number of different feature maps. LSTMs are slightly different, by their sequential nature compute is far heavier. Dropout is a good idea. Changing batch size of course proportionately changes compute time. Beyond a certain point for all models going deeper or wider did nothing to improve predictive ability on an unseen validation set, but of course worsened train/ validate time.

3 Difference between F1 and cross-entropy loss: Question 3

Describe a case where minimising the cross-entropy loss does not improve the F1-score.

First lets look at the equations:

$$F_1 = 2\frac{1}{\frac{1}{recall} + \frac{1}{precision}}$$

where,

$$precision = \frac{t_p}{t_p + f_p}$$

and,

$$recall = \frac{t_p}{t_p + f_n}$$

With of course t_p meaning true positive etc. Intuitively, if we guess everything in the world is a dog we will have great recall but pretty poor precision. The F1 gives a harmonic average to attempt to give us a reasonable balance as regards an accuracy metric, in short if we just say everything is a dog and most things are, we still don't have predictive ability but the harmonic average helps to illustrate this rather than a simple mean. In this example there are multiple classes (not all balanced, which adds its own complexity) but the metric takes a mean across the F1's.

If we look at cross-entropy...

$$\text{L(X,Y)} = \left(\tfrac{1}{N}\right) \sum_{i=1}^{N} y^{(i)} log(a(x^{(i)})) + (1 - y^{(i)}) log(1 - a(x^{(i)}))$$

As regards cross-entropy we are looking at outputs which give a probability of membership to a class for each data item. The test set will be one-hot encoded. F1 is a harmonic average of precision and recall. However F1 and cross-entropy are actually quite different. A couple of observations:

- Cross-entropy is differentiable, F1 is not.
- A perfect F1 does not imply a minimal cross-entropy. The loss can still reduce as the probabilities improve, thus aiding generalisation.
- Cross-entropy can improve at any stage of optimisation without improving F1.
- F1 can get worse as Cross-entropy improves, we can see this simply by imagining a couple of perverse examples.
- Here we are not taking F1 but a mean F1, which can alter things with imbalanced classes.
- We are doing mini-batch stochastic gradient descent on a not necessarily convex error surface. Thus, we are only approaching some local optimum as regards in expectation. On individual gradient descent's or even epochs we really have no guarantee of improving cross entropy and F1 at the same time.

Quick summary, as cross-entropy improves we would hope that on average so does F1. But there are no mathematical guarantees based upon individual descents over the error surface or even over epochs.

Verify this empirically using the code and dataset that was given in Question 2. Provide a graph that summarises your experiment. Finally describe briefly your experiment setup (architecture, specifications, etc.) and what the figure shows, and argue why this verifies your case. The argument should contain some explanation for the occurrence of these observations.

3.1 Experimental set up

Let us simplify everything to the 'bare bones'. First I created a highly imbalanced training example. The first class, versus the rest. Second I wanted to show that cross-entropy can improve when F1 goes down without worrying about generalisation errors. So I reported F1 and cross-entropy on the training set only. With this set up I created the simplest possible classifier (a softmax output) trained over many epochs, but I also checkpointed and saved the models at each epoch. I expected over time the model to be able to 'remember' all the training data and on average F1 to increase to 1 and cross-entropy to approach 0. But I also expected to see some bumps and isolated examples where cross-entropy reduced and F1 also reduced.

Figure 1, is perhaps not the most compelling figure in the world, (for example I could have chosen a perverse optimisation path doing maximal damage to F1 whilst still reducing cross-entropy), but we can see a few bumps along the optimisation road. F1 does reduce occasionally when cross-entropy reduces. To make this more compelling I performed a search on the times when this did happen. There were several, Table 5.

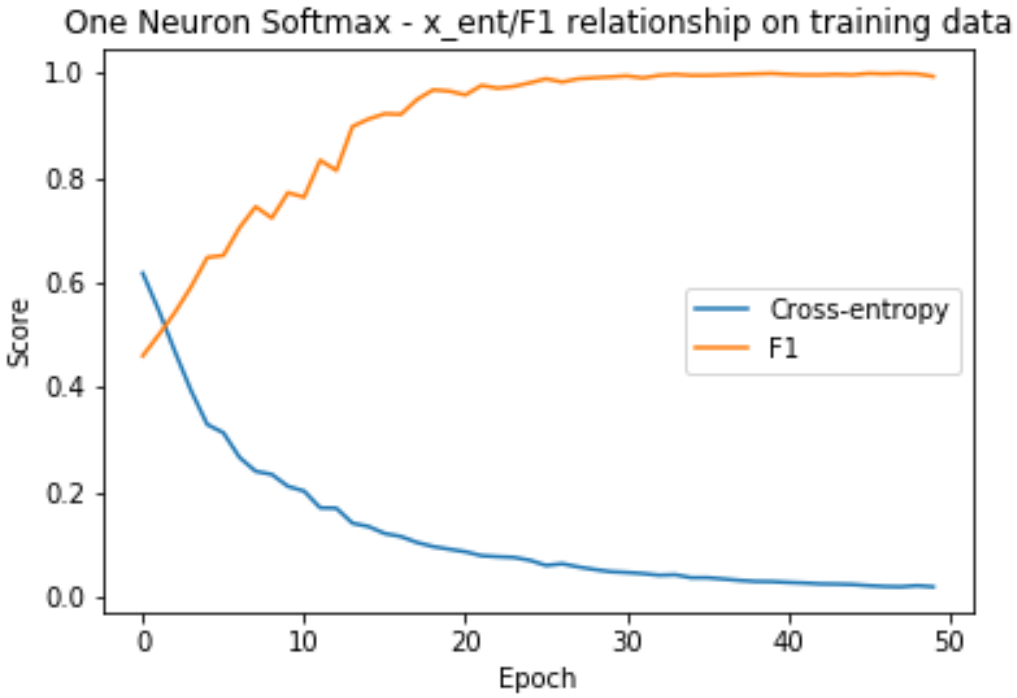

Figure 1: Cross Entropy v F1 each epoch

I will show the confusion matrices for the second of these examples, i.e. Epochs nine and ten. Here the F1 score goes down from 0.771 to 0.763, and the cross-entropy reduced from 0.212 to 0.203. As can be seen from the confusion matrix the validation prediction results have worsened despite a reduction in cross-entropy.

$$\begin{bmatrix} 263 & 12 \\ 393 & 8798 \end{bmatrix} \begin{bmatrix} 260 & 15 \\ 412 & 8779 \end{bmatrix} \tag{1}$$

Epoch	F1	Next F1	Xent	Next Xent
7	0.744505	0.722987	0.241088	0.235200
9	0.771243	0.762680	0.212442	0.203460
11	0.832911	0.813441	0.171571	0.170911
15	0.921382	0.920244	0.122700	0.117024
18	0.966877	0.964427	0.097835	0.092851
19	0.964427	0.957209	0.092851	0.087996
21	0.975371	0.970039	0.080485	0.078608
30	0.993480	0.989572	0.048592	0.046541
33	0.996242	0.994362	0.044247	0.038622
39	0.998121	0.996242	0.031236	0.029668
40	0.996242	0.995294	0.029668	0.028486
41	0.995294	0.995294	0.028486	0.026878
43	0.996228	0.995310	0.026482	0.025694
45	0.998121	0.997176	0.023514	0.021930
48	0.997186	0.992402	0.023560	0.021281

Table 5: F1 down/ Cross-Entropy down

4 Variational Autoenconders: Question 4

For this question, present a VAE setup with an alternate distribution (or distributions) used at any stage of the generational model pipeline, and derive the following:

- The loss function (c.f. the evidence lower bound [ELBO] which is based on the sampling and the regularization term) and/or gradient descent update rule.

- A practical way to implement the associated network's forward propagation (for purposes of training or inference).

4.1 Variational Autoencoder

The key reference for this question is 'Auto-encoding Variational Bayes', Kingma and Welling[2]. To give myself intuition and partly to prove I have grasped this! I have made a few points below, they are not strictly necessary for the question, but if there are any misconceptions I would welcome clarification.

- We take an autoencoder, (much of this is more about variational inference rather than autoencoding in my opinion, the autoencoder feels like the easiest bit).

- Instead of mapping data to a compressed vector we imagine our data is a sample from an underlying high dimensional distribution.
- We also imagine our data can be mapped to a simpler latent distribution, the encoder and decoder are simply neural nets or non-linear function mappings.
- We ask the encoder to map the sample to the latent distribution and the decoder to map back to the original.
- We cannot backprop through a random sample, so the cute part of the paper (at least in my opinion) is we perform a reparameterisation. Instead of mapping to a random variable we map to parameters defining the distribution of the random variable. For example if the latent distribution was a 20 dimensional gaussian we would require 40 neurons (for each mean and variance - assuming a diagonal covariance matrix).
- When we decode, we sample from that distribution first, then decode. In the paper they make life easy by sampling from a standard gaussian and subsequently transform.
- This enables us to backprop through these parameters while taking the randomness outside the neural net (at least in terms of gradients we care about).
- The constraint to the latent distribution is quite harsh. But this part seems key to making the network generative in some sensible way. In short, we can now move around that distribution and generate new samples 'close' to samples from the training distribution. If we didn't include this constraint, then moving around the latent space would not necessarily give intuitive results and indeed given enough flexibility in the neural net we could simply encode everything to an individual number, a bit like a dictionary lookup.
- The mapping to a constrained latent probability distribution is also a major difference with GANs where the discriminator merely 'polices' real verses fake. Here in say an mnist world. Moving around 7's in latent space we will presumably generate new 7's. GANs are different in simply making 'real' images from noise. Although they can of course be conditioned.
- The implications of original data distribution choice and latent distribution choice seem to be not entirely clear (at least to me). Anyway, we end up performing an optimisation (hence the variational solution), comprising two parts to the cost function. One a KL between the encoder output distribution and the latent distribution and the other the log likelihood of the decoder output under the data distribution. GANs have a different cost function which in the case of images gives sharper images but sometimes lacks variety (grabs modes). Whereas Variational Autoencoders are more blurry (I don't quite know if this is implicit in any cost function or simply because the chosen distributions are too 'smooth').

4.2 Back to the question-Cost function

From Kingma and Welling [2], we have a dataset $\mathbf{X} = (\mathbf{x}^i)_{i=1}^N$ consisting N, i.i.d samples and assume the data is generated by some random process involving an unobserved random variable $\mathbf{z}$. Hence our process comprises two steps, first $\mathbf{z}^i$ is generated by some prior distribution $p_{\theta^*}(\mathbf{z})$ and second a value $\mathbf{x}^i$ is generated by some conditional distribution $p_{\theta^*}(\mathbf{x}|\mathbf{z})$, where the prior and likelihood come from parametric families of distributions $p_\theta(\mathbf{z})$ and $p_\theta(\mathbf{x}|\mathbf{z})$, differentiable with respect to θ and $\mathbf{z}$. For the theory part of the paper the authors make no simplifying assumptions about the marginal or posterior probabilities. Hence this is more general than the standard example of the Variational Autoencoder where we use multi-variate Gaussians.

Also the authors assume that:

- The marginal likelihood $p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{z})p_\theta(\mathbf{x}|\mathbf{z})d\mathbf{z}$ is intractable.
- The posterior density $p_\theta(\mathbf{z}|\mathbf{x}) = p_\theta(\mathbf{x}|\mathbf{z})/p_{(\mathbf{x})}$ is also intractable.
- A very large dataset.

Hence we are ruling out approaches such as EM and sampling based approaches. The authors now use variational inference by introducing a model $q_\phi(\mathbf{z}|\mathbf{x})$ an approximation to the true posterior. The authors now derive a lower bound on the log-likelihood. i.e. $logp_\theta(\mathbf{x}^1, \ldots, \mathbf{x}^N) = \sum_{i=1}^N logp_\theta(\mathbf{x}^i)$.

$$\log \mathrm{p}_\theta(\mathbf{x}^i) = KL(q_\phi(\mathbf{z}|\mathbf{x}^i)||p_\theta(\mathbf{z}|\mathbf{x}^i)) + \mathcal{L}(\theta, \phi; \mathbf{x}^i)$$

Because the first item is a non-negative KL divergence, then we have a lower bound on the log-likelihood by simply working off the second term. After some manipulation (provided in the lecture notes and the paper, we arrive at.

$$\mathrm{L}(\theta, \phi; \mathbf{x}^i) = -KL(q_\phi(\mathbf{z}|\mathbf{x}^i)||p_\theta(\mathbf{z})) + \mathbf{E}_{q_\phi(\mathbf{z}|\mathbf{x}^i)} logp_\theta(\mathbf{x}^i|\mathbf{z})$$

We can now perform optimisation to maximise this lower bound, by minimising the KL and maximising the expected log-likelihood, which can be done by taking gradients with respect to the variational and generative parameters. However, the gradient with respect to the variational parameter is tricky and so the authors perform a further approximation involving a reparameterisation of the random variable $\tilde{\mathbf{z}} \sim q_\phi(\mathbf{z}|\mathbf{x})$ with a differentiable transformation. $\mathbf{z} = g_\phi(\epsilon|\mathbf{x}), \epsilon \sim p(\epsilon)$. By using a monte-carlo estimate the authors arrive at a new estimator of the lower bound.

$$\mathrm{L}^{\mathcal{B}}(\theta, \phi; \mathbf{x}^i) = -KL(q_\phi(\mathbf{z}|\mathbf{x}^i)||p_\theta(\mathbf{z})) + \frac{1}{L}\sum_{i=1}^{L} logp_\theta(\mathbf{x}^i|\mathbf{z}^{i,l}),$$

$$\text{where } \mathbf{z}^{i,l} = g_\phi(\epsilon^{i,l}, \mathbf{x}^i) \qquad and \qquad \epsilon^l \sim p(\epsilon)$$

Because of this reparameterisation if $\mathbf{z}$ is a continuous random variable, and $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$, then the key trick is that the random variable $\mathbf{z}$ can be re-expressed as a deterministic variable $\mathbf{z} = g_\phi(\mathbf{z}|\mathbf{x})$. There is still randomness, but for the purposes of the neural net it is taken outside and we are back-propping through deterministic parameters. In short:

$$\int q_\phi(\mathbf{z}|\mathbf{x}) f(\mathbf{z}) d\mathbf{z} = \int p(\epsilon) f(\mathbf{z}) d\epsilon = \int p(\epsilon) f(g_\phi(\epsilon, \mathbf{x})) d\epsilon \simeq \frac{1}{L}\sum_{l=1}^{L} f(g_\phi(\epsilon, \mathbf{x}^l))$$

Now for the standard variational autoencoder the authors take Gaussians giving an analytic solution to the KL. i.e. for the prior $p_\theta(\mathbf{z}) = \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I})$ and for the variational posterior they also use a multi-variate Gaussian (they also make life even easier by making it diagonal).

$$\log\ \mathrm{q}_\phi(\mathbf{z}|\mathbf{x}^i) = log\mathcal{N}(\mathbf{z}; \mu^i, \sigma^{2i}\mathbf{I})$$

For the sampling $\mathbf{z}^{i,l} = g_\theta(\mathbf{x}^i, \epsilon^l) = \mu^i + \sigma^i * \epsilon^l$, where $\epsilon^l \sim \mathcal{N}(0, 1)$

Thus, for the gaussian Variational Auto-Encoder the authors derived the estimate for the lower bound as, where z is sampled as above:

$$\mathrm{L}\,(\theta, \phi; \mathbf{x}^i) \simeq \frac{1}{2}\sum_{j=1}^{J}\left(1 + log((\sigma_j^i)^2)) - (\mu_j^i)^2) - (\sigma_j^i)^2\right) + \frac{1}{L}\sum_{l=1}^{L} logp_\theta(x^i|z^{i,l})$$

4.3 A new cost function

So I have made this a bit of a long story but we have:

- A prior latent distribution p($\mathbf{z}$).
- An assumed distribution of our data.

- A neural net encoder $q_\phi(\mathbf{z}|\mathbf{x})$ giving an estimate of the posterior density, but in reality providing the parameters for the distribution so we can sample the decoder whilst still being able to backprop.
- A decoder network $p_\theta(\mathbf{x}|\mathbf{z})$, which outputs into the distribution of the data.
- A cost function - which comprises a KL divergence between our encoder posterior and our latent distribution (which we wish to make as close to each other as possible) and a likelihood where we wish to maximise the likelihood of the decoder output under the data distribution.

Initially I wanted to make the output of the encoder a t-distribution as well as the latent variable. Thus, to do this my encoder would pass the parameters of the t. I would then sample and transform for the decoder distribution. My input distribution was to remain gaussian. In terms of the cost function I would need to calculate the KL divergence between the two multi-variate t-distributions. (The max likelihood part would remain unchanged). However my maths got messy!

So I chose the easy route. If we only change the input data distribution to a Bernoulli and thus the decoder output is also Bernoulli. Then I don't have to work out some new KL divergences and indeed the sampling remains exactly the same (I simply pass my gaussian parameters out of the encoder and sample for the decoder). The only thing that then changes is the max likelihood part of the cost function. This is an easy max-likelihood calculation.

$$\log(\mathrm{p}(\mathbf{x}|\mathbf{z})) = \sum_{i=1}^{D} x_i log(y_i) + (1 - x_i) log(1 - y_i)$$

Where y_i are the output from the decoder network. Apologies for the long winded nature of my answer. Given the actual change is really only a one liner I wanted to show I do actually understand what is going on.

5 Count the Parameters and understand why: Question 5

Lecture 2, provided examples of four autoencoder architectures in Keras. Determine how many trainable parameters are present for each of these architectures.

Discuss the manner in which the parameter count increases when adding further layers or computed features are added within each of these architectures. For your answer do not focus only on the code

but consider how each autoencoder differs.

5.1 Basic Auto-Encoder

The table for the basic auto-encoder parameter count is given below. Clearly this is an output from a Keras model summary. To give a little more colour, each layer is fully connected. So how do the parameters add up? Given an input dimension of 784, and a hidden layer with 32 neurons we will have 784*32+32 = 25120 parameters. (Because we have to add the 32 biases as well). To go to the next layer we will have 32*784+784 = 25,872 parameters (now we have 784 biases). Not much else to say I think.

Layer	Output Shape	Parameters
Input	(None, 784)	0
Dense	(None, 32)	25120
Dense	(None, 784)	25872
Total		50,992

Table 6: Auto-encoder

5.2 Deep Auto-Encoder

Again not much to say, 784*128+128=100480, 128*32+32=4128 etc, ie input layer of dimension 784, 128 hidden neurons and 128 biases. These are all dense layers so the calculations as given before.

Layer	Output Shape	Parameters
Input	(None, 784)	0
Dense	(None, 128)	100480
Dense	(None, 32)	4128
Dense	(None, 128)	4224
Dense	(None, 784)	101136
Total		209,968

Table 7: Deep Auto-encoder

5.3 Convolutional Auto-Encoder

A convolutional auto-encoder is a little different as regards hyper-parameters. The basic idea of a convolution is we have a kernel which passes over the image. This ties into the idea of receptive fields. Basically we are sharing weights and have no connections across distant pixels (in the example of images). Clearly there will be fewer weights than a dense layer.

When we pad the image or do max-pooling we are altering the image with padded zeros, or changing the dimension of the image by looking for maximum activations with respect to the kernel. Neither adds parameters for the neural network. What used to be called deconvolution, now seems to be called transposed convolution, which is just another convolution. Cropping again just crops an image and doesn't add parameters.

The basic calculation for this convolutional auto-encoder is to take the kernel height and width, multiply by the feature maps in the previous layer, multiply by the feature maps in this layer (for the weights) then add the biases (number of feature maps in the current layer).

Layer	Output Shape	Parameters	How!?
Input	(None, 28,28,1)	0	
ZeroPadding	(None, 32,32,1)	0	
Conv2d	(None, 32,32,16)	160	(3*3*1)*16+16
MaxPooling2d	(None, 16,16,16)	0	
Conv2d	(None, 16,16,8)	1160	(3*3*16)*8+8
MaxPooling2d	(None, 8,8,8)	0	
Conv2d	(None, 8,8,8)	584	(3*3*8)*8+8
MaxPooling2d	(None, 4,4,8)	0	
Conv2dTrans	(None, 8,8,8)	584	(3*3*8)*8+8
Conv2dTrans	(None, 16,16,16)	1168	(3*3*8)*16+16
Conv2dTrans	(None, 32,32,1)	145	(3*3*16)*1+1
Cropping2d	(None, 28,28,1)	0	
Total		3801	

Table 8: Convolutional Auto-encoder

5.4 Denoising Auto-Encoder

The only thing that is new in the denoising auto-encoder compared to the previous models is the addition of gaussian noise. This doesn't add any parameters. Which means all the calculations in the table below already been explained previously.

Layer	Output Shape	Parameters
Input	(None, 28,28,1)	0
GaussianNoise	(None, 28,28,1)	0
Conv2d	(None, 28,28,32)	320
MaxPooling2d	(None, 14,14,32)	0
Conv2d	(None, 14,14,32)	9248
MaxPooling2d	(None, 7,7,32)	0
Conv2dTrans	(None, 14,14,32)	9248
Conv2dTrans	(None, 28,28,1)	289
Total		19105

Table 9: Denoising Auto-encoder

6 Evaluate an existing model using basic operations: Question 6

Write a program to evaluate a simple feed-forward multi-layer neural network on the MNIST digits dataset (i.e., perform inference only given test data), using a pre-specified set of weights and bias terms that we provide (weights.zip).

You should assume the following network architecture for this question:

- Input layer: 784 values
- Hidden layer 1: 100 fully-connected neurons, with sigmoid activation
- Hidden layer 2: 50 fully-connected neurons, with ReLU activation
- Output layer: 10 fully-connected neurons, with softmax activation

Your answer should report the average accuracy your inference implementation achieves on the test and training sets of MNIST (in addition to the accompanying code used to obtain this result).Do not use higher-level abstractions, such as Keras layers that perform these low-level details for you.

The code is of course provided, but given there is so little I have included the basic changes in the document. I basically create a dictionary of weights which have been imported in the helper code given and make a multi-layer perceptron, trying not to stuff up dimensions.

```
weights = {
    'h1': tf.constant(layer_1_weights),
    'h2': tf.constant(layer_2_weights),
    'out': tf.constant(output_layer_weights)
}
biases = {
    'b1': tf.constant(layer_1_biases),
    'b2': tf.constant(layer_2_biases),
    'out': tf.constant(output_layer_biases)
}

def multilayer_perceptron(x):
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_1 = tf.nn.sigmoid(layer_1)

    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']),
                     biases['b2'])
    layer_2 = tf.nn.relu(layer_2)

    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer

# Construct model
```

```
logits = multilayer_perceptron(x)
out = tf.nn.softmax(logits)
```

```
with tf.name_scope('results'):
    correct_prediction = tf.equal(tf.argmax(y, 1),
                                  tf.argmax(out, 1))
    correct_prediction = tf.cast(correct_prediction, tf.float32)
    accuracy = tf.reduce_mean(correct_prediction)

# run the model
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
    print('Train accuracy:{}'.format(sess.run(accuracy,
            feed_dict={x: mnist.train.images,
            y: mnist.train.labels})))
    print('Test accuracy:{}'.format(sess.run(accuracy,
            feed_dict={x: mnist.test.images,
            y: mnist.test.labels})))
```

Within a tensorflow session I then test accuracy on the train and test set. I ended up with 0.973 for the training set and 0.965 for the test set.

7 Making Deep Learning models more efficient: Question 7

In Lecture 3, we discussed the resource constraints that extend from deep learning referencing in particular: energy, computation and memory. For two of these constraints, describe how the severity of the constraint is tied to network architecture design choices.

The question is quite open ended and obviously an active area of research. So I will try to narrow down the scope through specific intuitive examples.

First lets start with the premise that network architecture dominates predictive performance. Given that convolutional neural networks show superior performance for visual recognition and recurrent networks for sequence prediction etc, I would argue that this is reasonably uncontroversial.

I will concentrate on fully connected feed forward networks compared to CNN's to further narrow scope. In question 5, we already answered the question as regards the number of parameters for different networks (albeit in an auto-encoding context). We could see clearly that in a fully connected network for example the number of weights between layer L-1 and layer L was $m_{l-1}m_l$, where m_l was the number of neurons in layer l. With a deep network a reasonable approximation would be W^L weights. These of course all need to be stored, hence a Deep MLP has a heavy duty memory requirement, which constrains it

when used on hardware with a small footprint.

If we contrast this with a CNN. We showed in question 5 that we had far fewer hyper-parameters to train and described how weights are shared with many weights being zero (if one translates to an MLP context it is like having a prior of zero on weights between distant pixels). The memory requirements for a CNN are thus far less. However, when we performed the tests on training and inference time, we found a CNN had a far heavier load. This is logical, we are passing a kernel across every segment of the image, for all channels, repeatedly, depending upon how many feature maps we desire. The compute is $O(NCd^2HW)$, (maps, channels, height, width and dimension of kernel).

So we can see clear trade-offs in terms of architecture design versus memory and compute. LSTM's by their sequential nature would have a heavy compute load again, but thats another story. I have concentrated on memory and compute, these (all else being equal) would often equate to energy, but not necessarily. For example, data stored outside working memory that requires continual requests to pull may have a heavy energy load.

In this lecture, we also discussed how deep model architectures once trained are often described as "inefficient" by those people considering resource constraints. What is meant by this?

What is meant by deep model architectures being inefficient is the fact that some massive trained models can be compressed drastically for the purposes of inference without damaging performance very much at all. Why is this? The model may have millions of weights, yet many may actually be close to zero. The aggregate activation outputs from many neurons may also contribute next to nothing. In some respects it appears we over-engineer our models at training time. Indeed, if we didn't use dropout or some form of regularisation, or early stopping - then left unchecked epoch after epoch, our model would memorise our training data, but have little ability to generalise. Yet even a properly constructed model with good generalisation ability still has many redundant weights and neurons.

Provide a specific example of inefficiency in the model representation observed popular architectures (like AlexNet etc.) once they are used.

There are plenty of papers out there on this. For the purposes of answering this part of the question I refer to 'Tensorizing Neural Networks', Novikov et al. [3] and 'Deep Compression: Compressing Deep Neural Networks with pruning, trained quantization and huffman coding', Song Han, et al. [1]. In these two papers, on the ImageNet dataset:

- The authors [1] compressed AlexNet by 35x, from 240MB to 6.9MB, with-

out loss of accuracy.

- Reduced the size of VGG-16 by 49x from 552MB to 11.3MB, again with no loss of accuracy.
- Benchmarked on CPU, GPU and mobile GPU, the compressed network had 3x to 4x layerwise speedup and 3x to 7x better energy efficiency.
- In [3], for the Very Deep VGG networks the authors compressed the dense weight matrix of a fully-connected layer up to 200,000 times and the whole network up to 7 times.

I would argue this presents strong evidence that these huge models have clear inefficiencies and there is an on-going story as regards improving this and porting them to smaller devices such as mobile devices.

8 Node pruning - How and what implications: Question 8

In Lecture 3, we discussed node pruning as a method employed to reduce/reshape resource consumption of deep models. Describe how node pruning is performed and how it improves resource efficiency.

If one starts with a visualisation of a deep fully connected network where we already estabilised in the previous question that we have many redundancies. i.e. weights close to zero and activations with outputs that don't really impact the final result. Then the next obvious step is to remove connections and neurons in some way. We showed earlier the compute load of a CNN and the memory load of fully connected layers. Clearly if we can remove connections or neurons without impacting predictive ability we have resource gains, either computational, or memory or both. Currently there appears to be a whole zoo of techniques for trying this, but a couple of obvious examples that appear to stand out:

- Autoencode a fully connected layer. Between a fully connected layer we basically have a matrix of weights. Reduce the dimension in some way. Say SVD and insert a lower dimensional layer (looks like a linear autoencoder or PCA basically).
- Sparse coding - Basically a similar philosophy with a dictionary lookup.

On a personal note the idea of training a model first and then pruning nodes in some heuristic way afterward doesn't feel perfect. I preferred the idea that energy or efficiency is embedded into the initial optimisation problem. My analogy with a simpler world is that it reminded me of doing some L2 regularised

optimisation and then naively removing features with small weights. This may not be quite right as the aggregate of these many small weight features could actually be relevant.

Provide one heuristic/approach to determining if a node can be pruned or not. Often after the node pruning process additional training is performed, what does this accomplish? Why do you think accuracy improves when this is done?

A heuristic might be to look at the trained weights coming into the neuron, or the activations coming out. Cull the neuron if we are below a certain level. Again I would highlight I am not entirely convinced that this is the best approach and perhaps the culling should be embedded in the training process.

Additional training is performed because by removing neurons and the associated connections we have a different architecture, with no reason to believe the existing trained weights are now 'optimal'. There are all sorts of entanglements going on in neural nets, if we retrain the new architecture we can subtly adjust remaining weights and perhaps improve our cost. At least by doing so we have everything to gain by retraining and nothing to lose.

References

[1] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[2] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[3] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. Tensorizing neural networks. In *Advances in Neural Information Processing Systems*, pages 442–450, 2015.