Deep Neural Networks (DNN) Assignment

Nicholas Drake
University of Oxford

13-17 March 2023

Total Number of Pages: 15

1 a) What is the dimension of $\frac{dy}{d\mathbf{w}}$?

I have $y = \sum_{i=1}^{N} w_i x_i^2$ which is a scalar-valued function of Nvariables $\mathbf{x}$ each with its own weight making $\mathbf{w}$ a vector of length N also. The dimensions of this vector can either be vertical $N \times 1$ or horizontal $1 \times N$ depending upon whether I prefer the Jacobian or gradient layout. Given the context of backpropagation and gradient descent it is perhaps more natural to choose the latter.

1 b) Compute an analytical expression for $\frac{dy}{d\mathbf{w}}$

As $\mathbf{w}$ is a vector I can define each partial derivative $\frac{dy}{dw_i} = \frac{d(w_i \times x_i^2)}{dw_i}$ where I have substituted in for y and used the sum rule and the fact that the derivative of a constant is zero. Furthermore, since x_i^2 is a constant with respect to w_i the partial derivative becomes $\frac{dy}{dw_i} = x_i^2$ Expressed as a vector this can be written as

$$\frac{dy}{d\mathbf{w}} = \begin{pmatrix} x_1^2 \\ x_2^2 \\ \vdots \\ x_N^2 \end{pmatrix}$$

Note, here I have written the partial derivatives as $\frac{dy}{dw_i}$ in an effort to match the question and the lecture slides rather than $\frac{\delta y}{\delta w_i}$ etc.

1 c) Given the backpropagated error $\frac{dE}{dy}$

(i) What is the dimensionality of $\frac{dE}{dy}$

I am not told what the cost function is, but whether mean squared error, or cross-entropy loss it will be some sort of scalar output function that I can minimize. Therefore the dimensionality of $\frac{dE}{dy}$ depends entirely upon the dimension of the output y which as the weighted sum of squares is also a scalar and therefore so is $\frac{dE}{dy}$. Thus I know its dimension is 1

(ii) How would you compute $\frac{dE}{d\mathbf{w}}$?

I can use the chain rule such that

$$\frac{dE}{d\mathbf{w}} = \frac{dE}{dy}\frac{dy}{d\mathbf{w}}$$

Substituting in our analytical expression for $\frac{dy}{d\mathbf{w}}$ I get

$$\frac{dE}{d\mathbf{w}} = \begin{pmatrix} \frac{dE}{dy}x_1^2 \\ \frac{dE}{dy}x_2^2 \\ \vdots \\ \frac{dE}{dy}x_N^2 \end{pmatrix}$$

(iii) Given the gradient how would you update the weights?

To update the weights I can use an optimization algorithm like gradient descent. This involves subtracting the product of the backpropagated gradient multiplied by the learning rate r. The intuition behind this approach is that the gradient, in our case the derivative $\frac{dE}{d\mathbf{w}}$, represents the maximum uphill direction so in order to go downhill I should go in the opposite direction.

$$\mathbf{w}' = \mathbf{w} - r\frac{dE}{d\mathbf{w}}$$

Substituting and simplifying I have

$$\mathbf{w}' = \begin{pmatrix} w_1' \\ w_2' \\ \vdots \\ w_n' \end{pmatrix} = \begin{pmatrix} w_1 - r\frac{dE}{dw_1} \\ w_2 - r\frac{dE}{dw_2} \\ \vdots \\ w_n - r\frac{dE}{w_n} \end{pmatrix}$$

The learning rate r represents the step size and a trade-off between a smaller r and slower convergence versus a larger r and potentially overshooting the optimal solution. For this reason a number of optimizers like Adam vary the step size to take account of 1. Momentum where the updates are smoother as they take account of previous gradient calculations 2. Adaptive learning rates considerations where the parameters with historically large gradients receive smaller step sizes, while parameters with small gradients receive larger learning rates.

Thus, if I use Adam[1] the weight update becomes

$$\mathbf{w}' = \mathbf{w} - r\frac{\mathbf{m}'}{\sqrt{\mathbf{v}'} + \epsilon}$$

where I have replaced the gradient $\frac{dE}{d\mathbf{w}}$ with the standardized moment plus a small constant ϵ for numerical stabilisation preventing numerical underflow and division by zero.

This is calculated by maintaining two exponentially decaying moving averages: 1. One of the mean (first-order moment) $\mathbf{m}$ where the exponential decay rate ρ_m determines how much weight should be put on the previous estimate $\mathbf{m}$ vs the newly calculated gradient $\frac{dE}{d\mathbf{w}}$ and the denominator $1 - \rho_m$ corrects the bias.

$$\mathbf{m}' = \frac{\rho_m \mathbf{m} + (1 - \rho_m)\frac{dE}{d\mathbf{w}}}{1 - \rho_m}$$

2. One of the variance (second-order moment) $\mathbf{v}$ where the exponential decay rate ρ_v determines how much weight should be put on the previous estimate $\mathbf{v}$ vs the newly calculated squared gradient $\frac{dE}{d\mathbf{w}}$(technically the Hadamard product) and the denominator $1 - \rho_v$ corrects the bias

$$\mathbf{v}' = \frac{\rho_v \mathbf{v} + (1 - \rho_v)\frac{dE}{d\mathbf{w}} \odot \frac{dE}{d\mathbf{w}}}{1 - \rho_v}$$

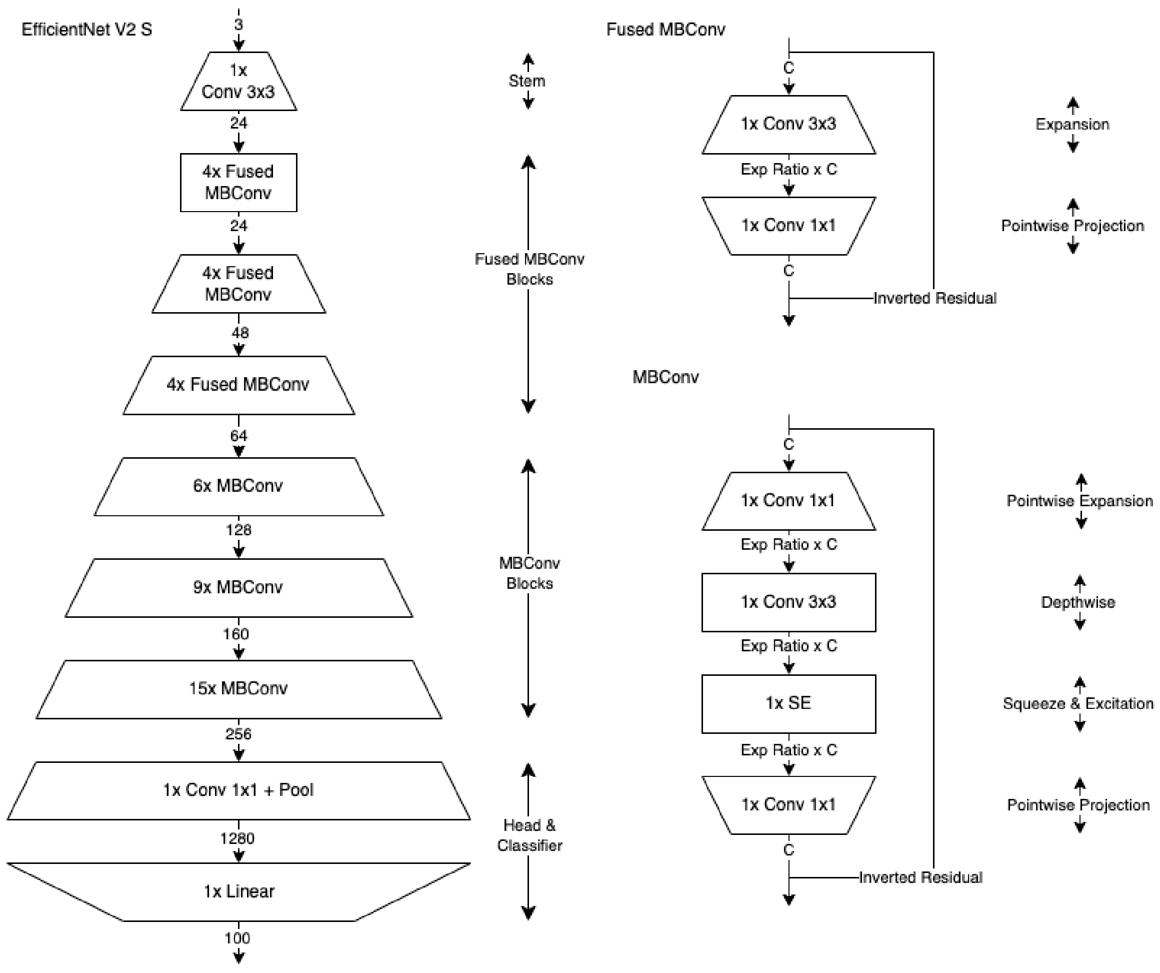
I have explored gradient descent and Adam but there are many other possible optimization algorithms with adaptive learning rates such as RMSProp and AdaDelta or with Momentum like Nesterov[1].

2 In this question the challenge is to design, implement and train a CNN architecture for the CIFAR-100 dataset

a) The architecture of the network as a diagram

I use the PyTorch implementation of EfficientNet V2 Small[2]. Note there are slight differences in the implementation compared to the original paper[3], in particular the Fused MBConv's in the PyTorch implementation do not have the squeeze-excitation layers. This and other aspects of the architecture are discussed in c).

The diagram shows the EfficientNet architecture on the left from an initial stem to the layers of Fused MBConv blocks, and then the MBConv blocks before finally a head and classifier. The FusedMBConv blocks and MBConv blocks are illustrated on the right hand side. The number between each block indicates the channel size where C is the input number of channels and Exp Ratio is the expansion ratio set in the blocks (note this is only relevant for the first of the MBConv/Fused MBConv in each block).



b) A description on how you prepared and used the datasets

I prepare the data in the transform_data function. This involves

1. Downloading the CIFAR-100 train and test datasets
2. Splitting the train dataset 0.8:0.2 into train and validation sets
3. Apply transforms to train and test. CIFAR-100 images are 32x32 but I resize these to 224x224[3] and normalize to the ImageNet mean and standard values[3]. For the train and validation transforms I also flip, rotate and resize crop.
4. I use PyTorch's DataLoader iterator to shuffle and batch into groups of 64, the largest batch size that will fit on the GPU.

c) A description on the main architectural elements in your network as well as the motivations for your design choices

I chose to use the EfficientNet V2 architecture[3] which is the basis of the SOTA CIFAR-100 performance[4] of EffNet-L2[5].

Initially, I tried implementing EfficientNet V2 from scratch (see Appendix of the attached Python notebook). This included a generic implementation of MBConv/Fused MBConv blocks and customizable layers to allow config driven creation of EfficientNet V2 S, M or L. However, when trained from scratch on CIFAR-100 the model achieved only 72% validation test accuracy. I also trained training the PyTorch model[2] from scratch and achieved a similar accuracy validating my implementation but still far from the SOTA benchmark. This was achieved by using pretrained weights from ImageNet and fine-tuning on

CIFAR-100 which meant I couldn't use my implementation and instead needed to use the off the shelf implementation from PyTorch[2].

The EfficientNet paper is focused on model scaling with a single compound efficient across depth, width and resolution[6]. They develop their baseline architecture by using multi-objective neural architecture search that optimizes both accuracy and FLOPS[6]. The result is an EfficientNet architecture based upon mobile inverted bottleneck MBConv modules[7] which combine point-wise and depth-wise convolutions with squeeze and excitation blocks[8].

- Separable convolutions: Full convolutions generate each output point from input across the kernel k^2 and across all channels d_i. Separable convolutions factorise these into two parts: a depth-wise convolution across the kernel and a point-wise convolution across all channels. Their combined effect is almost as the same as a full convolution but with a significantly reduced computational cost $\approx k^2$[7] which for a 3×3 kernel could be roughly 9x.

- Linear bottlenecks: I can capture the key relationships, or so-called manifold of interest, in a low-dimensional subspace through linear bottlenecks (non-linear perform less well as they collapse some of the subspace e.g. for ReLU the part less than 0)

- Squeeze and excitation blocks: Squeeze layers provide a global summary per channel of the spatial information in a layer. Excitation blocks learn to filter the relevant information with a self-gating mechanism[8].

EfficientNet V2 also adds to the EfficientNet-B0-7 architectures[6] Fused MBConv modules which helps address the bottleneck of depth-wise convolutions in early layers[3] by replacing the depth-wise Conv 3x3 and expansion Conv 1x1 in MBConv with a single regular Conv 3x3. In the PyTorch implementation they also remove the Squeeze and Excitation[2] as shown in the previous section's diagram. This results in the EfficientNet V2 S architecture shown where the first few layers are FusedMBConv whilst the later layers are MBConv. I use the small architecture rather than the medium or large because they 1. Took up too much space on the GPU 2. Took longer to train and 3. Required images to be scaled up to larger sizes and therefore small batch sizes and more compute time.

Although I used the pretrained weights from ImageNet as CIFAR-100 only has 100 classification categories compared to ImageNet's 1000 I replace the final classifier layer with a linear layer with just 100 outputs in cifar100_classifier function.

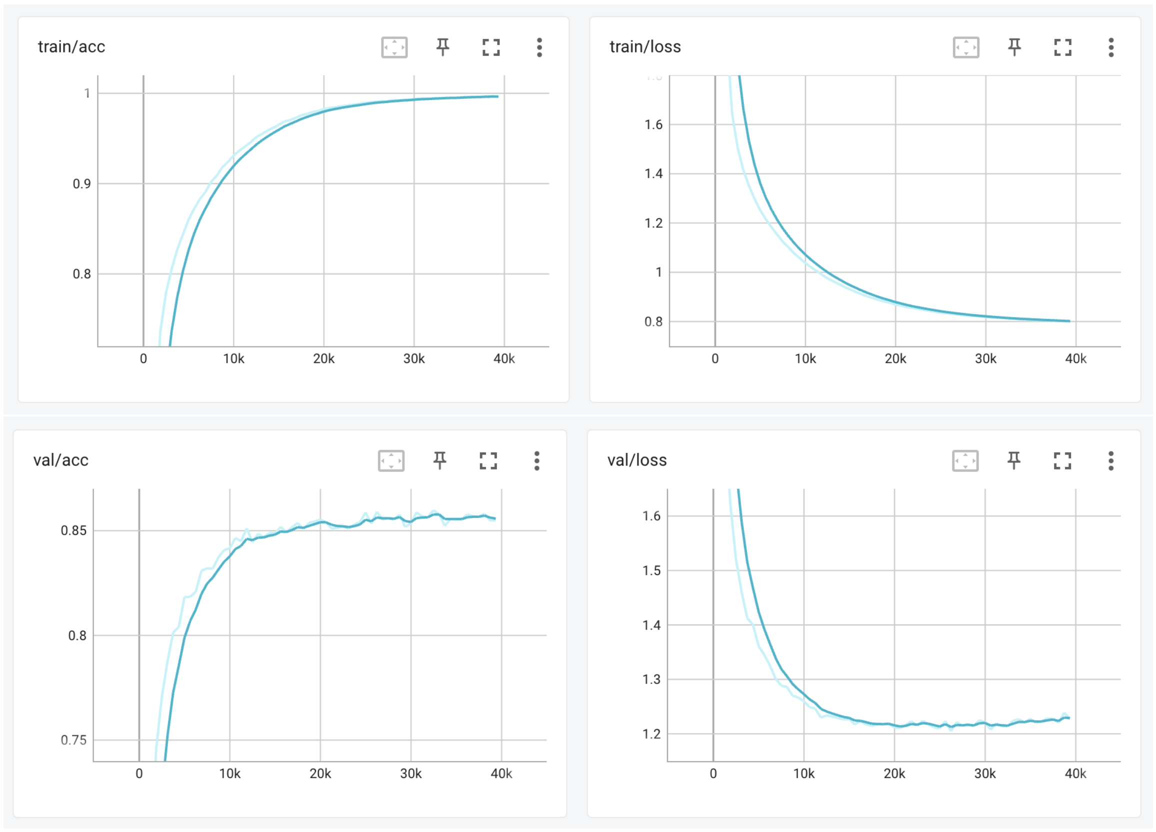
d) The final accuracy on test set and train and validation accuracy and error curves

I do two separate runs

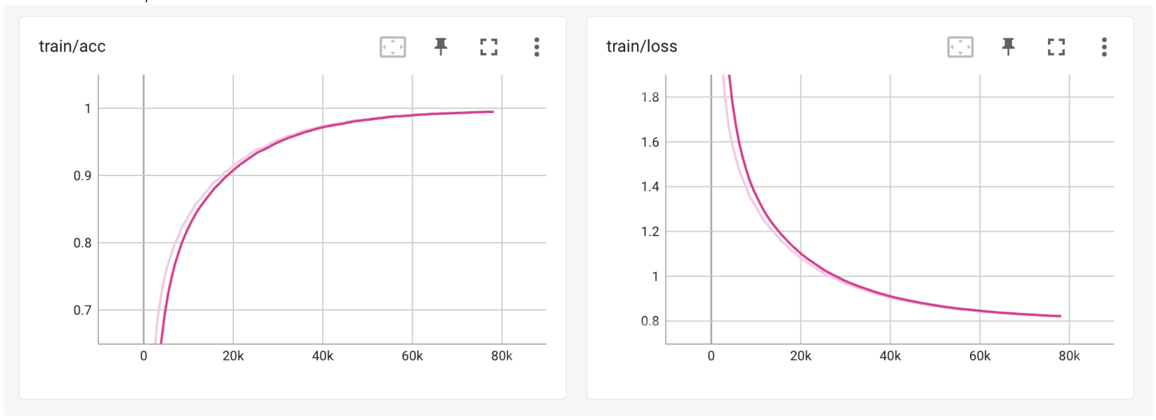
1. I do an 80-20% train-validation split of the training data using early stopping after 10 epochs to save the highest validation accuracy model of 85.98%. On some runs, the early stopping was not triggered unless I tried a higher number of max epochs (in this case 100) but despite paying for the Google Colab+ subscription and a lot of compute time I found that my runs would sometimes fail because there is a 12 hour limit or because Colab disconnects if you do not respond.

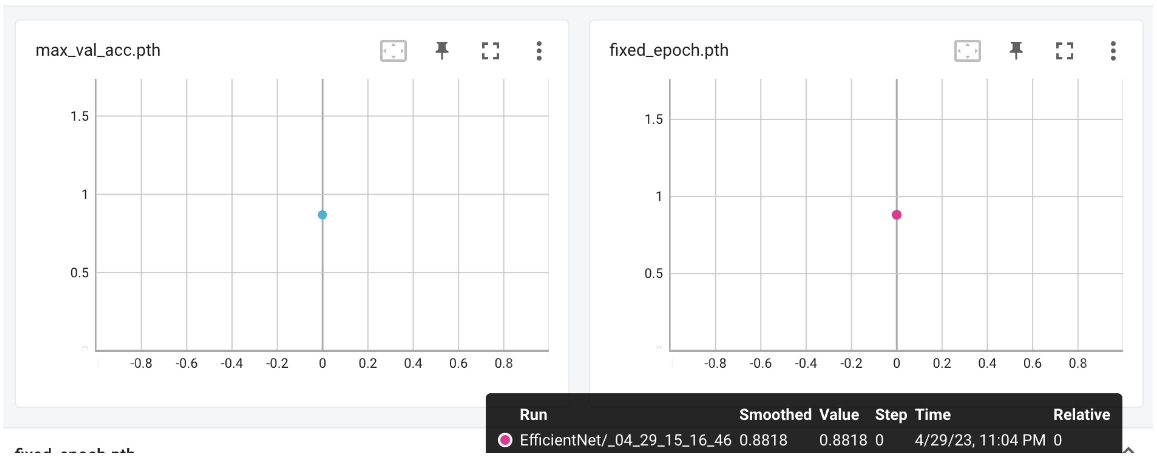
2. I also do a 100% train split with no validation, no early stopping and a fixed 100 epochs in order to maximise the amount of training data I have for our test performance. I choose 100 epochs as I have more training data to train and because from our experiments and the validation accuracy curves for our SAM implementation[5] generalise well and are unlikely to over-fit.

For the 80-20% train-validation run these are the train and validation loss and accuracy curves where the final accuracy on the test set was 86.91% (see max_val_acc.pth Tensorboard graph). You will want to filter on EfficientNet/_04_29_12_29_33



For the 100% train run these are the train loss and accuracy curves where the final accuracy on the test set was 88.18% (see fixed_epoch.pth). Note there are no validation graphs for this 100% train test run. Also the learning rate is fixed for Adam although the calculated step is not. You will want to filter on EfficientNet/_04_29_15_16_46





Run	Smoothed	Value	Step	Time	Relative
● EfficientNet/_04_29_15_16_46	0.8818	0.8818	0	4/29/23, 11:04 PM	0

e) A brief summary of the experiments you conducted to obtain the hyper-parameters (e.g. learning rate, batch size, dropout rates etc.) you chose for your best model

Choosing the best weights

- From scratch: I tried training from scratch both my custom implementation of EfficientNet and the off the shelf version. However, both achieved roughly 72% validation accuracy, well off the SOTA mark.
- Freezing base model: I did try training with a two-part training process initially freezing all the pretrained weights and just training the classifier first for 25 epochs and then training the whole model later but I found this actually achieved poorer results than training the whole model directly.
- Freezing batch normalization: Following the example of SAM[5] we freeze the batch normalization layers which yields a marginal improvement in test performance over not freezing those layers (88.1% vs 88.18%)
- Dropout on base model: Given that the pretrained weights are already refined, I assumed that a low dropout rate as recommended[3], on the base model would be appropriate but in practice found that a dropout rate of 0.4 on both the classifier and the main model performed a percentage or two better than any other alternatives. Perhaps this is because the base model has to be retrained quite significantly because of the low image resolution of the resized 224x224 images (from 32x32)[5]
- Random initialization of the classification layer: I did try random initializing the weights in the classification layer but this did not seem to make any difference in terms of the final performance of the model and if anything it trained less smoothly so I did not use this in the best performing run

Choosing the best optimizer

- RMSProp: I tried training with ImageNet pretrained models copying the hyper-parameters from EfficientNet V2[3] which included RMSProp optimizer with decay 0.9, momentum 0.9, batch norm momentum 0.99 and weight decay 1e-5 with the smaller initial learning rate 1e-3 they use scaled for the fact that I was using batches of size 64 8 times smaller than their batches of 512. These smaller batches were necessary to fit the data in on the GPU. I also used a separate scheduler with cosine annealing, however nonetheless I found it very difficult to train where the training loss exploded to NaN. I tried to mitigate this by smaller learning rates but this made the learning too slow. I tried warming up the learning rate but this also did not help much.
- Adam: I tried default Adam with learning rate 1e-3 but perhaps because of my smaller batch size of 64 it was very volatile and did not train well. Changing the learning rate to 1e-5 I found Adam trained well with greater than 80% validation accuracy. Adding label smoothing to the cross entropy loss function also helped a lot[5]. It acts as a form of regularization by replacing the hard 0 and 1 k classification targets with target of $\frac{\epsilon}{k-1}$ and $1-\epsilon$ respectively[1] preventing the model over-training in pursuit of hard targets whilst still providing enough of a signal. I use the default $\epsilon = 0.1$ which works well.

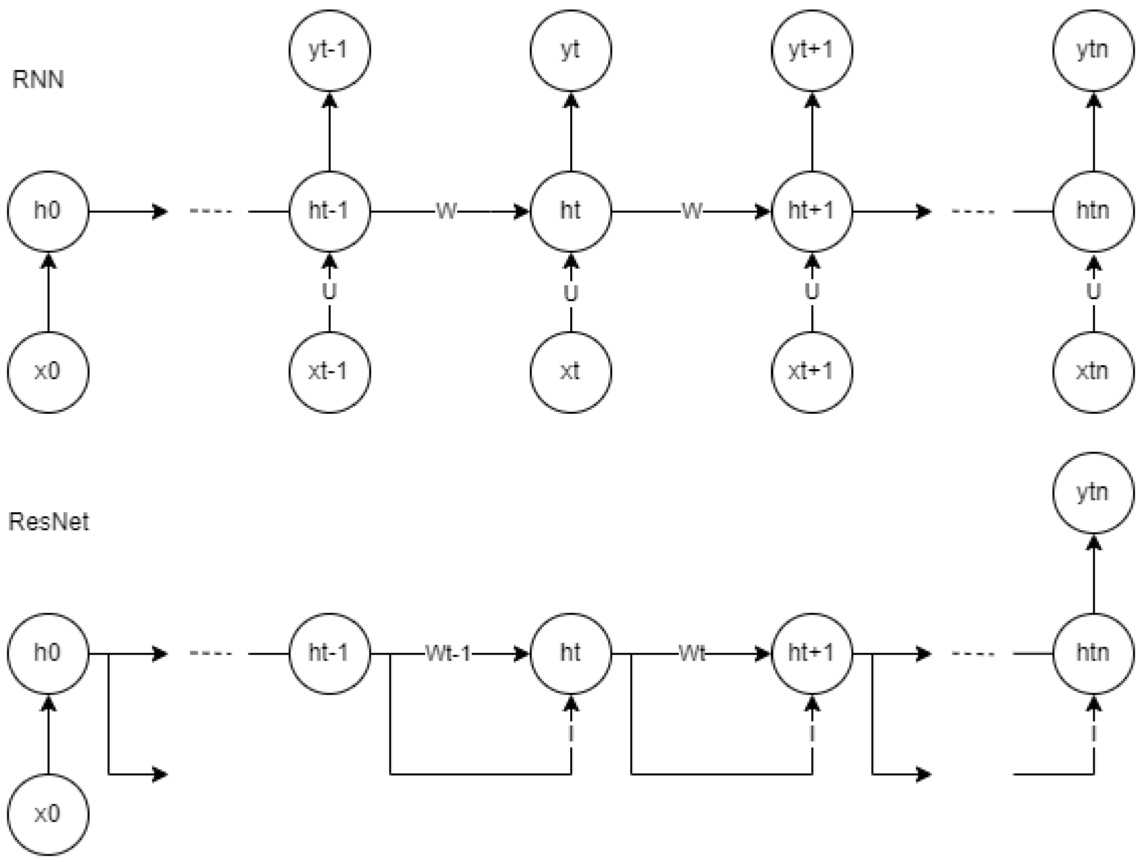
- SGD: I also tried with momentum 0.9 and cosine annealing rate decay[5] but as with RMSProp found it very difficult to train with either NaN losses or extremely small learning rates.
- SAM: I also tried using Sharpness-Aware Minimization[5] as in the SOTA performing[4] implementation. SAM involves a two step optimization process where 1. I calculate $\hat{\epsilon}(\mathbf{w})$ which is the local neighbourhood ρ maximum around a given point $\mathbf{w}$ 2. I then perform a second forward backward pass around that point to calculate the gradients but crucially I apply those gradients to the original weights. The result is that I perform gradient descent on effectively the worst performing point in the local neighbourhood which incentivizes finding flatter regions of the loss space. I implement a generic SAM implementation which can be seen in the PyTorch code based upon the publicly available TensorFlow code[9] and a PyTorch implementation[10]. The implementation involves a two-step forward backward pass using a closure to implement the second SAM pass. As recommended in the SAM paper I use a default $\rho = 0.05$ where we constrain the local maximum point $||\epsilon||_2 \leq \rho$ to have a norm less than ρ. Adding SAM helps significantly resulting in our best performing model on the validation set with more than 86% test accuracy
- ASAM: My implementation of SAM also supports Adaptive SAM[11] which involves re-scaling the neighbour hood region ρ based upon the parameter size but despite using the recommended, significantly larger, $\rho = 1$ I was not able to get good performance out of the model at all, perhaps further hyper-parameter tuning is required.

f) The source code (Colab notebook files)

See attached cifar100.ipynb

3 Sketch a clear diagram showing the main architectural differences between an unrolled RNN and a ResNet. List and explain each difference, referring to your diagram

Recurrent Neural Networks (RNN) and Residual Networks (ResNet) are two common types of neural networks. Despite quite different origins: ResNets were developed to help prevent the vanishing gradients problem across layers through skip connections and RNNs to model sequences, they have a number of architectural similarities. For example, both take an input $\mathbf{x}_0$ and pass state between layers until producing an output $\mathbf{y}_{t_n}$. In fact, it can be argued that ‘ResNets with shared weights can be reformulated into the form of a recurrent system’[12]



However, a number of differences remain including:

- Hidden state inputs: RNNs (illustrated in unrolled form in the diagram) take the hidden state of the previous step $\mathbf{h}_{t-1}$ and the input sequence (a flexible number) $\mathbf{x}_t$ as inputs. There are also variations that take the output of each hidden state $\mathbf{y}_t$ as the input to the next state, ones where the is no $\mathbf{x}_t$ input except for $\mathbf{x}_0$, and even ones with skip connections itself. ResNet also has two inputs but both are the previous hidden state: one an untouched residual connection and one a transformed version.

- Parameter sharing: The RNN has shared weights for its hidden-to-hidden connections, in our example, parameterized by $\mathbf{W}$ and input-to-hidden connections parameterized by $\mathbf{U}$, formally known as the stationarity assumption[1]. ResNet in contrast does not usually share weights $\mathbf{W}_t$ with separately learned weights for each layer. Furthermore, the skip connections typically have no weights at all with just an identity function.

- Hidden state outputs: RNNs often produce an output after each step $\mathbf{y}_t$, whilst ResNet does not. Some RNNs produce an output only after the final layer however, or repeatedly until some special token is outputted signifying the end of the sequence.

- Nonlinear function: A typically RNN recurrence function might be of the form

  $$\mathbf{h}_t = \tanh\left(\mathbf{b} + \mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t\right)$$

  where I have an affine function of the previous hidden state $\mathbf{h}_{t-1}$ and the current input $\mathbf{x}_t$ combined with a nonlinear activation function such as tanh. In contrast, ResNet has residual learning every few stacked layers. If I assume two affine layers with an activation function between as in the original ResNet paper[13] then I have

  $$\mathbf{h}_t = \mathbf{b}_{t_2} + \mathbf{W}_{t_2}(\sigma(\mathbf{b}_{t_1} + \mathbf{W}_{t_1}\mathbf{h}_{t-1})) + \mathbf{h}_{t-1}$$

  where σ is a nonlinear function such as ReLU and the residual connection $+\mathbf{h}_{t-1}$ is performed by a shortcut connection and element-wise addition. This identity connection might be adjusted with an additional weight $\mathbf{U}\mathbf{h}_{t-1}$ to map the shortcut connection to the dimensions match. Thus the two hidden states are both updated with a combination of affine and non-linear activation functions the main difference is that residual building blocks tend to have several weight layers and RNNs take in additional input $\mathbf{x}_t$ rather than the hidden state.

4 Given the description of an AlexNet-style network, compute the output size, number of weights and biases of each layer.

There are three types of layer in the proposed network

- Convolutional layers
  - Output size = Floor[14] of (Input width - Kernel width + 2 x Padding) / Stride + 1 for width and height multiplied by the Number of Output Channels
  - Number of weights = Kernel width x Kernel height x Number of input channels x Number of Output Channels
  - Number of biases = Number of Output Channels
- Max Pool layers
  - Output size = Floor[15] of (Input width - Pooling width)/ Stride + 1 for width and height multiplied by the Number of Output Channels
  - Number of weights = 0 because it just calculates the maximum and so has no learnable parameters
  - Number of biases = 0 because it just calculates the maximum and so has no learnable parameters.
- Fully Connected layers
  - Output size = Number of neurons
  - Number of weights = Input size × Output size
  - Number of biases = Number of neurons

	Layer output size	No. of weights	No. of biases
Layer 1: Input	$256 \times 256 \times 3$	0	0
Layer 2: Conv-1	$\frac{256-7+2\times0}{2}+1 = 125.5 \longrightarrow (125 \times 125 \times 64)$	$7 \times 7 \times 3 \times 64 = 9408$	64
Layer 3: MaxPool-1	$\frac{125-3}{2}+1 = 62 \longrightarrow (62 \times 62 \times 64)$	0	0
Layer 4: Conv-2	$\frac{62-5+2\times2}{1}+1 = 62 \longrightarrow (62 \times 62 \times 256)$	$5 \times 5 \times 64 \times 256 = 409600$	256
Layer 5: MaxPool-2	$\frac{62-3}{2}+1 = 30.5 \longrightarrow (30 \times 30 \times 256)$	0	0
Layer 6: Conv-3	$\frac{30-3+2\times1}{1}+1 = 30 \longrightarrow (30 \times 30 \times 512)$	$3 \times 3 \times 256 \times 512 = 1179648$	512
Layer 7: Conv-4	$\frac{30-3+2\times1}{1}+1 = 30 \longrightarrow (30 \times 30 \times 512)$	$3 \times 3 \times 512 \times 512 = 2359296$	512
Layer 8: Conv-5	$\frac{30-3+2\times1}{1}+1 = 30 \longrightarrow (30 \times 30 \times 512)$	$3 \times 3 \times 512 \times 512 = 2359296$	512
Layer 9: MaxPool-3	$\frac{30-3}{2}+1 = 14.5 \longrightarrow (14 \times 14 \times 512)$	0	0
Layer 10: FC-1	4096	$(14 \times 14 \times 512) \times 4096 = 411041792$	4096
Layer 11: FC-2	4096	$4096 \times 4096 = 16777216$	4096
Layer 12: FC-3	1000	$4096 \times 1000 = 4096000$	1000

5 You want to train a simple deep network using the following loss function: $\mathcal{L} = \sqrt{\frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{N}}$. The network starts training but after a few iterations, the loss turns to not-a-number (NaN).

a) What could the problem be? Show how you arrived at this conclusion

Following the hint to consider the backpropagation stage I calculate the partial derivative of the loss function

$$\frac{\delta \mathcal{L}}{\delta \hat{y}_i} = \frac{\delta \sqrt{\frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{N}}}{\delta \hat{y}_i}$$

Using the chain rule I differentiate the square root first and then the MSE term inside

$$\frac{\delta\mathcal{L}}{\delta\hat{y}_i} = \frac{1}{2}\frac{\sum_{i=1}^{N}(y_i-\hat{y}_i)}{N}^{-\frac{1}{2}} \times \frac{\delta\frac{\sum_{i=1}^{N}(y_i-\hat{y}_i)^2}{N}}{\delta\hat{y}_i} = \frac{1}{2\sqrt{\frac{\sum_{i=1}^{N}(y_i-\hat{y}_i)^2}{N}}} \times \frac{\delta\frac{\sum_{i=1}^{N}(y_i-\hat{y}_i)^2}{N}}{\delta\hat{y}_i}$$

I calculate the partial derivative of the MSE by the chain rule and the sum rule on the summation

$$\frac{\delta\frac{\sum_{i=1}^{N}(y_i-\hat{y}_i)^2}{N}}{\delta\hat{y}_i} = \frac{\sum_{i=1}^{N} 2\times(y_i-\hat{y}_i)\times\frac{\delta(y_i-\hat{y}_i)}{\delta\hat{y}_i}}{N}$$

Substituting in $\frac{\delta(y_i-\hat{y}_i)}{\delta\hat{y}_i} = -1$ I again use the sum rule and the fact that y_i is a constant with respect to $\hat{y}_i$ and so its derivative is 0

$$\frac{\delta\frac{\sum_{i=1}^{N}(y_i-\hat{y}_i)^2}{N}}{\delta\hat{y}_i} = -\frac{2}{N}\sum_{i=1}^{N}(y_i-\hat{y}_i)$$

Finally, substituting back into the loss partial derivative I have

$$\frac{\delta\mathcal{L}}{\delta\hat{y}_i} = \frac{1}{2\sqrt{\frac{\sum_{i=1}^{N}(y_i-\hat{y}_i)^2}{N}}} \times -\frac{2}{N}\sum_{i=1}^{N}(y_i-\hat{y}_i) = -\frac{\sum_{i=1}^{N}(y_i-\hat{y}_i)}{N\sqrt{\frac{\sum_{i=1}^{N}(y_i-\hat{y}_i)^2}{N}}} = -\frac{\sum_{i=1}^{N}(y_i-\hat{y}_i)}{\sqrt{N}\sqrt{\sum_{i=1}^{N}(y_i-\hat{y}_i)^2}}$$

The issue is that the $\sqrt{\sum_{i=1}^{N}(y_i-\hat{y}_i)^2}$ term could underflow to zero resulting in a divide by zero error and NaN.

b) What simple change would you make to the loss function to prevent this issue?

I could tweak the loss function by adding a small positive constant ϵ such as e^{-8} or e^{-10}

$$\mathcal{L} = \sqrt{\frac{\sum_{i=1}^{N}(y_i-\hat{y}_i)^2+\epsilon}{N}}$$

The partial derivative of a constant is again 0 but by the chain rule the ϵ is preserved inside the square root preventing division by zero ever being possible (note the sum of squares is always greater or equal to zero)

$$\frac{\delta\mathcal{L}}{\delta\hat{y}_i} = -\frac{\sum_{i=1}^{N}(y_i-\hat{y}_i)}{\sqrt{N}\sqrt{\sum_{i=1}^{N}(y_i-\hat{y}_i)^2+\epsilon}}$$

6 A deep neural network is trained to diagnose a disease that occurs in 1 in 20 individuals. The model achieves 96% accuracy across 100 test subjects.

a) Assuming the disease occurs in the test subjects exactly at the average rate, calculate the worst-case precision for the model

Precision is defined as $\frac{TP}{TP+FP}$ where TP is the number of true positives and FP is the number of false positives

Accuracy is defined as $\frac{TP+TN}{TP+TN+FP+FN}$ where TN is the number of true negatives and FN is the number of false negatives

The model achieves 96% test accuracy and occurs at exactly the average rate of 1 in 20 in the test subjects meaning 5 of the 100 have the disease.

The worst-case precision would be that all four inaccuracies are FP with only one correct diagnosis which means that the precision $= \frac{TP}{TP+FP} = \frac{1}{1+4} = \frac{1}{5} = 0.2$ i.e. 20%

b) Now calculate the best-case precision for the model.

The best-case precision for the model is that all four inaccuracies are FN with a single correct prediction TP which means that the precision $= \frac{TP}{TP+FP} = \frac{1}{1+0} = 1$ i.e. 100%

c) Would you say this is a "good" model for the use case? Should it be deployed?

96% seems like a very high accuracy, but this is misleading because of the class imbalance and relative rarity of the disease where only 1 in 20 people has it. In fact a naive test that always came out negative would achieve a 95% accuracy but clearly this would not be a useful diagnostic tool.

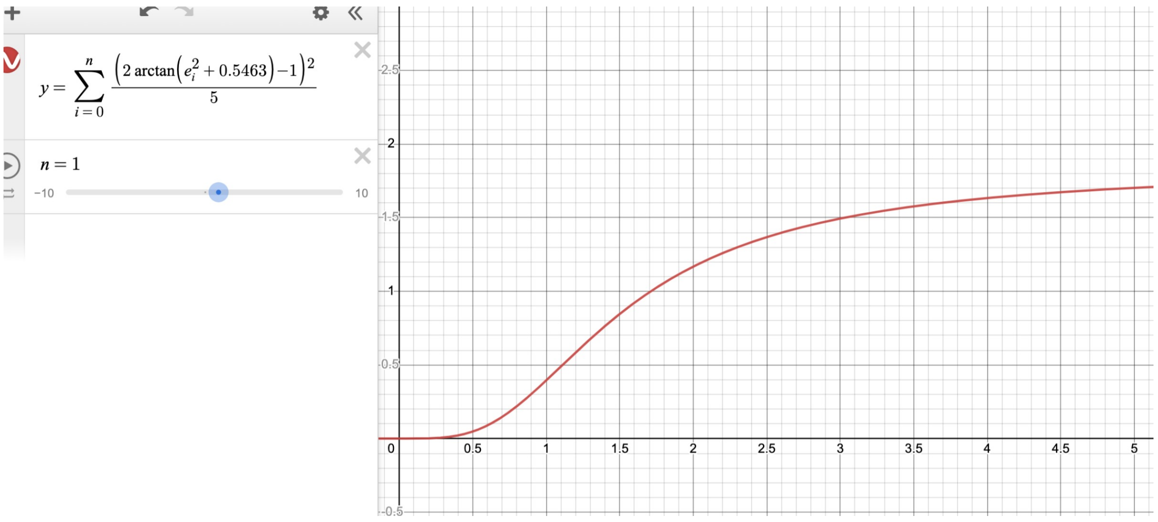
I have already considered precision which is the percentage of positive diagnoses which are correct. Of course, telling someone they have a disease when they do not is far from ideal so this is an important consideration.

But when it comes to diagnosing disease it is even more important that individuals with the disease were not missed. This is captured by the concept of recall which is defined as $= \frac{TP}{TP+FN}$. As I did with precision I can calculate the worst-case recall where all the errors are FN which gives $= \frac{TP}{TP+FN} = \frac{1}{1+4} = 0.2$ i.e. 20%. Conversely the best case recall is all the errors are FP so the recall $= \frac{TP}{TP+FN} = \frac{5}{5+0} = 1$ i.e. 100%.

Thus there is a trade-off between precision and recall where I would want to investigate further to see whether the inaccuracies are more likely to be FN or FP. As discussed with disease diagnose the preference would be for the latter and the higher recall and lower precision this would imply. However, ultimately given the low class incidence 96% accuracy is not high enough and I would say on balance that is not a good model and it should not be deployed.

7 Consider the following loss function $\mathcal{L} = \sum_i \frac{(2\arctan(\epsilon_i^2+0.5463)-1)^2}{5}$ where $\epsilon_i = y_i - \hat{y}_i$ is the error for sample i, computed as the difference between the ground truth label y_i and the prediction $\hat{y}_i$

a) Draw the graph of the loss function for $\epsilon > 0$


$$y = \sum_{i=0}^{n} \frac{\left(2\arctan\left(e_i^2 + 0.5463\right) - 1\right)^2}{5}$$

$n = 1$

b) Say you are trying to train a deep network on a dataset with noisy samples and relatively few training labels. Is this a good loss to use? Explain your reasoning

Suitable loss function

- Higher error, higher loss: The loss increases as the sample error increases, and it goes to zero when the error is zero.
- Symmetry: It is symmetrical which other loss functions like cross-entropy loss is not. In particular, because we square the ϵ term $y_i - haty_i$ has the same loss as $-(y_i - \hat{y}_i)$. In the context of a binary classifier this can mean balanced penalties for false positives and false negatives which may be desirable.
- Smoothness: The curve is smooth meaning that it is differentiable at all points, and more stable gradients.

- Arctan dampening: With noisy samples you want to choose a loss function that is less sensitive to outliers. For example, Mean Squared Errors puts more weight on outliers (with no limit) which in this case might not be appropriate. The arctan function with its S-shape has the effect of dampening outliers.
- It also dampens correct (or almost correct) answers focusing on those that are slightly wrong which can also be beneficial. This leads to a similar effect as focal loss[16] in putting less weight on easy examples that are predicted correctly.

Not suitable loss function

- Saturation: Typically the choice of loss function is tightly coupled with the type of output unit[1]. For example when using MLE and the negative log likelihood (i.e. cross-entropy) it can work very well with exponential based activation functions like the sigmoid, as the log in the cost function undoes the exponential of the sigmoid resulting in a linear gradient which provides strong guidance. Although the arctan dampening can be advantageous as discussed above it also means there is likely to be insufficient learning gradient.
- Class imbalance: Small sample sizes are more sensitive to class imbalance so perhaps putting more weight on less frequently occurring classes could be helpful as is done in weighted cross entropy loss.
- Regularization: There is no direct regularization term to encourage smaller weights for example.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[2] Efficientnet, pytorch.

[3] Mingxing Tan and Quoc V. Le. Efficientnetv2: Smaller models and faster training. *CoRR*, abs/2104.00298, 2021.

[4] Image classification sota on cifar-100.

[5] Pierre Foret, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. Sharpness-aware minimization for efficiently improving generalization, 2021.

[6] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. 2020.

[7] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. 2019.

[8] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu. Squeeze-and-excitation networks. 2019.

[9] Tensorflow implementation of sam.

[10] Pytorch implementation of sam.

[11] Jungmin Kwon, Jeongseop Kim, Hyunseo Park, and In Kwon Choi. Asam: Adaptive sharpness-aware minimization for scale-invariant learning of deep neural networks, 2021.

[12] Qianli Liao and Tomaso A. Poggio. Bridging the gaps between residual learning, recurrent neural networks and visual cortex. *CoRR*, abs/1604.03640, 2016.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. 2015.

[14] Conv2d. pytorch 2.0 documentation.

[15] Maxpool2d. pytorch 2.0 documentation.

[16] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection, 2018.