MACHINE LEARNING COURSEWORK

Part 1 – Building a Classifier

Development/experimentation strategy

Research online and the lectures given showed that CNNs are good for this task, so an initial CNN was created. Quick tweaking on the first few epochs resulted in a network with 2 convolution layers and 2 linear layers with ReLUs in-between. Running this for 30 epochs (a few minutes) gave the following:

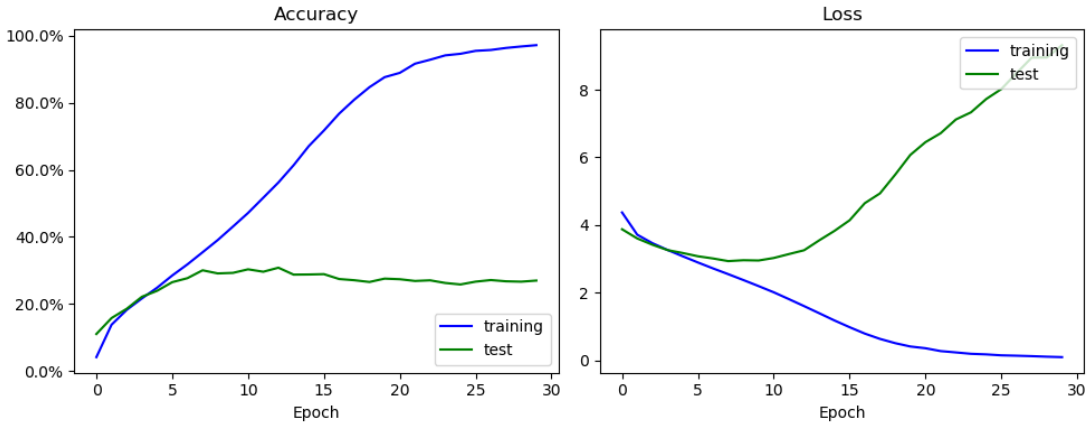


Fig. 1a – Accuracy/loss from the initial CNN

While the training accuracy was good, this was from overfitting, so the testing accuracy was mediocre. Generalising the network (through regularisation) was therefore imperative. Dropout was first attempted using multiple rates for different layers. Literature online suggested that convolution layers should have low dropout at best, with later linear layers having higher dropout (except for the last layer(s)). Unfortunately, in all attempts the training speed was orders of magnitude slower. It also seemed that if we waited until convergence it would still be worse. L2 normalisation (weight_decay in the optimiser) had a similar effect, as dropout and L2 are very similar.

Batch normalisation and tensor normalisation showed some improvement in terms of regularisation. With tensor normalisation we changed the input from [0, 1] to [-1, 1], the idea being that we have negative weights to work with. Batch normalisation also appeared to speedup the training of the NN (at least within epochs).

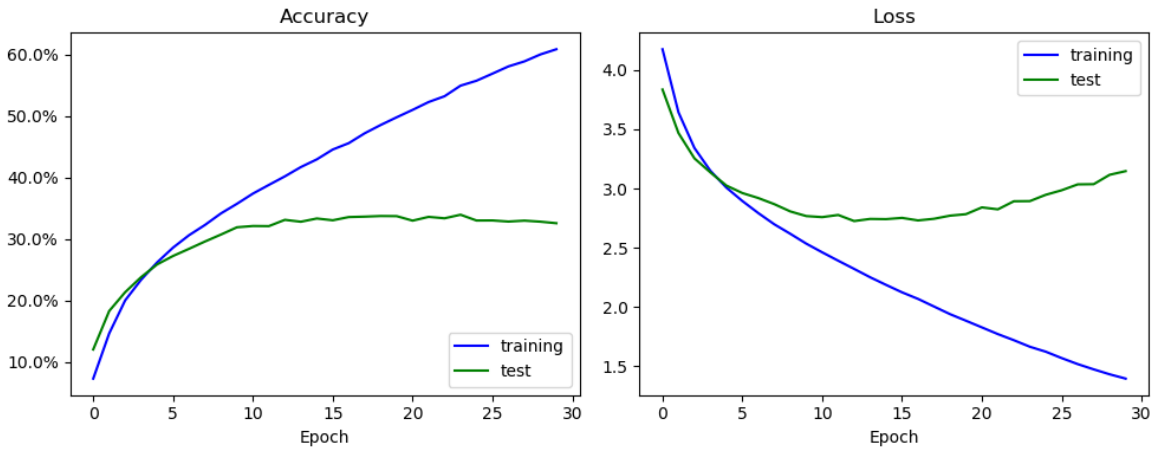


Fig 1b. Initial results of attempts to reduce overfitting

Figure 1b shows that while we reduced the rate of overfitting, it is still there just on a slower timescale, and our testing accuracy has not improved by much. Data augmentation was the final explicit regularisation technique used and randomly flipping images horizontally had a decent impact. Additional or alternate augmentation (eg. random rotations) didn't improve things any more.

Aside from explicit regularisation, the design of the architecture is also important. We see from fig. 1b that we are memorising the training dataset to nearly 100% accuracy. Our initial CNN only consisted of a few convolutional and linear layers (with batch normalisation added between convolutional layers). The first tweaking was mostly blind experimentation to avoid the model collapsing to 1% accuracy. So the next step was to try and improve the CNN architecture itself.

We could do simple experimentation with kernel sizes, number of layers, etc. but this was mostly guessing. Our aim with CNNs is to recognise low scale features then aggregate them into higher features before passing them through linear layers. Through experimentation the format converged upon was a small initial kernel, a larger kernel that is strided and dilated (to recognise large features), then a final large kernel to aggregate this. 3 linear layers were then used after this.

Pooling was added between convolution layers. This allowed us to quickly reduce the network complexity (for generalisation/running time) while also keeping the important details. Maxpool seemed most effective for the first convolution layer to recognise the most important low-level features. Later convolutional layers didn't benefit from maxpool, but did from avgpool, as our aim was aggregation. Obviously the various parameters were experimented on to try and find ones that worked best.

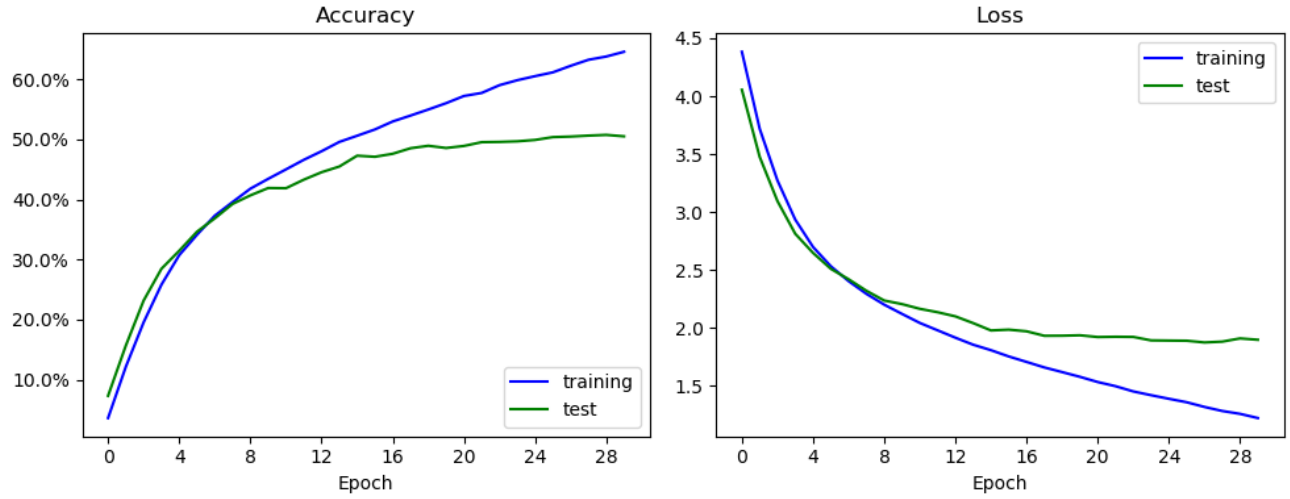


Fig. 1c Results after more regularisation/CNN generalisation

We see that this network performs considerably better in testing accuracy and overfitting. The initial regularisation attempted appears to work much better with the improved CNN, and we get a test accuracy of above 50%. Our overfitting rate has been reduced, but if we run for more epochs we do still end up overfitting.

<u>Minor experiments</u>

Some of the more mundane things experimented on were:

- Batch sizes – This increased/decreased the epoch length, but also changed the training rate. High batch sizes also caused wobbles in accuracy and were wholly unsuitable. The initial value given seemed optimal.
- Training on the entire training dataset each epoch seemed to give better results than on a subset (although epochs took longer)
- Using LeakyReLUs, ELUs, etc. instead of ReLUs did not seem to make a difference
- Applying momentum (0.9) to the optimiser had a significant effect, although changing the learning rate did nothing

Final accuracy push

Our work and experimentation reduced the overfitting and has given us good accuracy, but accuracy is the only metric we are interested in (for this assignment). By reducing the number of linear layers from 3 to 2, massively boosting some of the training parameters (mainly the hidden linear layer), and running for more epochs we were able to get another 5-10% accuracy. This did increase the training time and resulted in terrible overfitting again, but the increase in accuracy was significant. Ultimately the generalisation efforts were critical to developing a good architecture which we could exploit at the end to increase accuracy as much as possible.

Final results

Best testing accuracy: 58.7%

Test accuracy and plot loss:

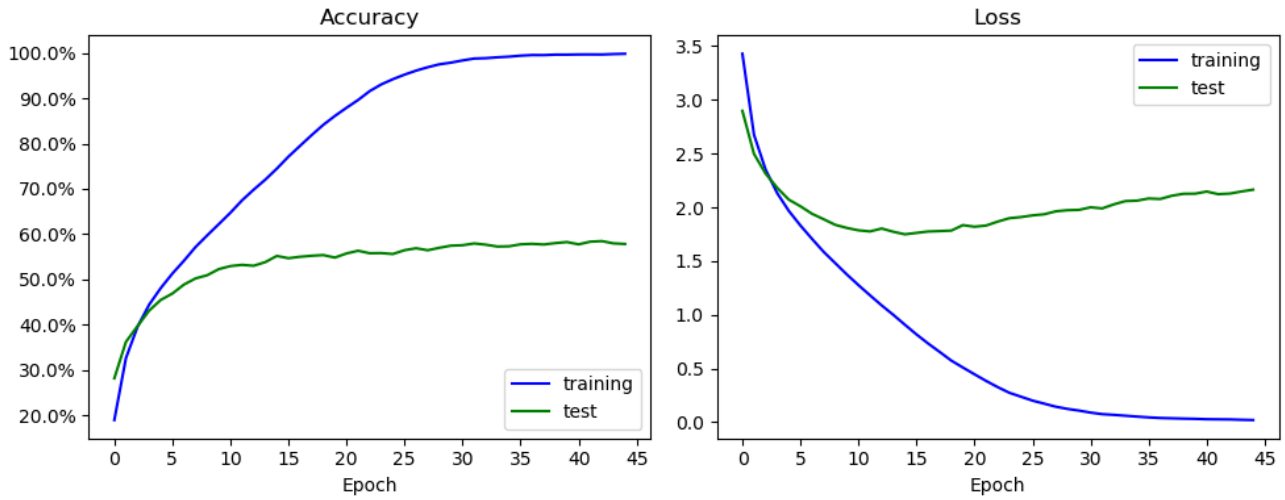


Fig. 1d – Final test plot

Part 2 – Generating a Pegasus

Initial analysis

Both autoencoders and GANs are used as generative models and as we were given an autoencoder I investigated if GANs (and extensions) could be an alternative. The idea being that we could construct a generator function where the discriminator cannot distinguish if the image is a horse or bird (hence a pegasus). However, it is likely that in practice this model would be incredibly difficult to train and susceptible to collapse. For example, if we just generated noise then the discriminator could not discern anything. We could modify this to try and ensure that we still have horse/bird features (eg. by combining the original discriminator with one that discriminates between generated images), but again this process seems difficult to train. One advantage of GANs (compared to autoencoders) is that they smooth the output less, so are less blurred.

I therefore decided against GANs and worked on autoencoders. The first thing to do was to separate the dataset into only horses/birds and train on just horses to see how good our autoencoder is at encoding and decoding a specific image class.

Fig. 2a – Initial autoencoder results for horses

From 2a we see that our generated images are of such poor quality that they cannot be used to generate a pegasus. We must therefore improve the quality of the generated images before we consider combining them. The classifier from part 1 was brought in to try and improve this, but many modifications were needed. For starters we effectively need to be able to reverse each layer of the classifier to create our decoder. For convolutional layers we can use ConvTranspose2d to effectively 'deconvolve' the layer identically. Sometimes tweaks were needed such as output_padding to retain the same tensor size. The pooling used is not invertible so we would have to upscale (leading to blurring). This had to be dropped and compensated for with additional convolutional layers. Finally we had to make sure our latent space appropriately sized, otherwise we would end up encoding images of horses directly (or encoding no information).

Fig. 2b – CNN Autoencoder results for horses

From 2b we see that most of our generated images have the main features of a horse (head, body, legs, tail). For images of such small resolution the blurriness is not that bad. The problem is that all the images are very similar (brown horse on green background) and we have only been training on the horse dataset. When we train on 2 classes (horses/birds) we get much worse results.

Fig. 2c – Images of 'horses' when training the CNN autoencoder on birds + horses

The encodings of both datasets are interfering with themselves, so we want a way to separate this out. Fortunately VAEs can be used to normalise the encodings so that hopefully our input distributions will occupy distinct regions (and we can still interpolate between them). For this I used the functions from the lecture example on github to augment the existing autoencoder. It was even more imperative that the latent space was not too large, as otherwise the result was a featureless blur.

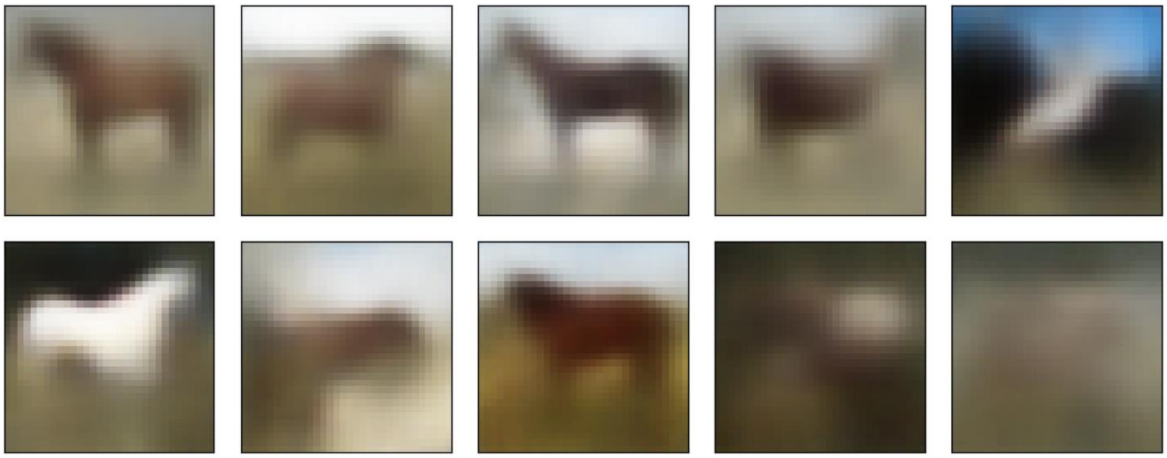


Fig. 2d – Images of horses when training the CNN VAE on birds + horses

We now start to see horses again in 2d. Admittedly the images are blurrier and of worse quality compared to 2b, but we are now training on multiple classes and our horses are still recognisable. Alongside the CNN parameters the optimiser/loss function was always played around with by tweaking the amount the KL divergence and cross entropy loss contributes towards the total loss. I also tried modifying this ratio over time, but ultimately this had little success.

<u>Creating a pegasus</u>

This task proved exceptionally difficult; the CIFAR-10 dataset is poor as most images of birds are stationary and have complicated backgrounds. Instead I sampled aeroplanes as they had clear wings attached to the bodies, although there were still many images that were wholly unsuitable. For combining the images, I used the standard multiplication of the encodings by different amounts, experimenting with ranges from 0-1. The problem was that interpolated images either contained a horse, a grey smear (aeroplane), or an unrecognisable blur.

Overall I went for a larger horse bias compared to aeroplanes as the horse has more distinctive features than a simple wing. The next page shows this, with the bottom left image being the best pegasus found.

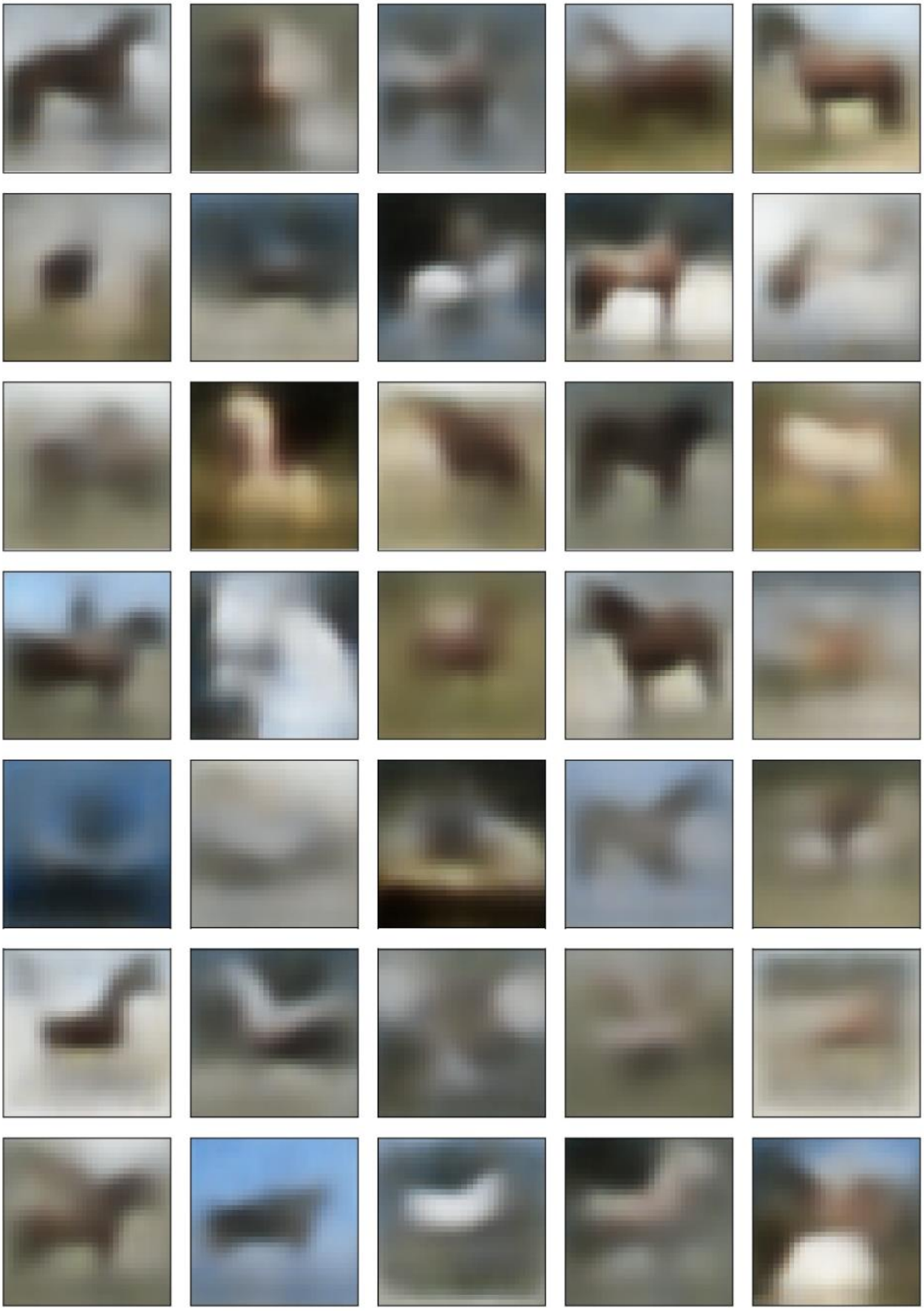
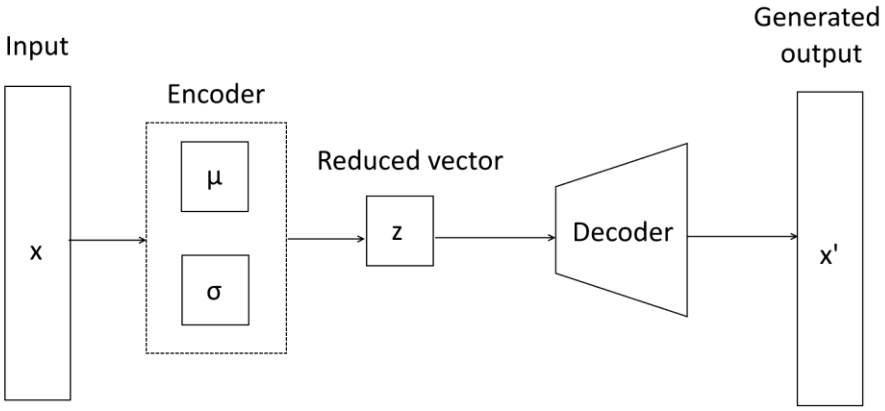


Fig. 2e – VAE combining random horses and aeroplanes

Final generative model architecture

Since we are using a VAE our architecture is well known. We take the input images and convert them to a reduced latent space based on the mean and standard deviation. We then use a decoder to try and reconstruct this image from the latent space.

NN architecture:



We then encode a random horse/aeroplane, combine them via summation (and multiplied), before decoding the tensor created.

Best pegasus created

After many horses and blurry images this is the best result I had: