Problem 1

The paper presents a framework to improve image classification through self-supervised algorithm that trains the model to identify key features in images by contrasting augmented (perturbed) versions of these images. In supervised image classification it is costly to label images one by one, so there is a practical limit to which we can improve the model by just adding labeled data.

SimCLR bypasses this problem by creating augmented versions of unlabelled images, whereby augmentations refer to various perturbations on images such as cropping and resizing, rotation, and re-coloring. SimCLR pass these augmented images through an encoder (which may be transferred from a pre-trained model suh as RESNET-50) and an additional non-linear projection head to project onto some latent space. It then uses NT-XENT loss (which I will go in detail in Question 2 so will skip over the detail here) to train the encoder network parameters so that the network will produce representations where pairs of augmented images from the same parent image will have higher cosine similarity score and images with different parent image will have low similarity score.The algorithm is self-supervised because we don't need to know the lable of any of the original images and only need to keep track of which pairs of augmented images came from the same parent image. With the encoder network parameters retrained, SimCLR gets rid of the projection head and fine-tune the encoder network parameters with few labelled images.

SimCLR framework outperformed previous self-supervised frameworks, and with more layers in the network achieves almost the same accuracy as supervised algorithms at around 76.5% top-1 accuracy. The result also shows that self-supervised contrastive learning framework benefits more from larger batch sizes and more training steps compared to supervised learning. A more subtle point in the paper was that although we discard the projection head in our final task to classify new test data, the inclusion of projection head in the training process improves performance substantially. Finally, data augmentation is also an important choice to make while training. The paper finds that random cropping combined with random color distrotion perfroms reasonably well. Color distortion is especially important because without it the neural network will exploit color distributions of images which is not a reliable way to classify images.

I think the results of the paper is interesting because it seems to be proposing a framework that is closer to how human brain processes images. The brain does not have perfect storage of every single details of an object, but rather just few defining features that allow it to identify the object under different circumstances and distortions. It is interesting that by mimicking this feature of human perception SimCLR is able to achieve good classification results without the need for lots of labeled data. While some have commented that many of the machine learning frameworks in use right now are essentially black boxes, developing framework like this with a very well defined learning goals instead of simply exploring various non-linearities and adding more and more layers to the network seem to be the way to balance performance and intepretability.

Problem 2

The loss described in the paper is NT-Xent (Normalized tempreature-scaled cross entropy loss). The form of the loss for a batch of size N is as follow:

$$L = \frac{1}{2N}\sum_{k=1}^{N}[\ell(2k-1,2k)+\ell(2k,2k-1)]$$

$$\text{Where: } \ell_{ij} = -log\frac{exp(sim(z_i,z_j)/\tau)}{\sum_{k=1}^{2N}\mathbb{1}_{[k\neq i]}exp(sim(z_i,z_k)/\tau)}$$

$$\text{And: } sim(u,v) = \frac{-u^Tv}{\|u\|\,\|v\|}$$

In the above we have a batch of size N, and we apply augment functions to N images to produce two augmented images for each and hence end up with 2N images. Each image therefore has one possible pair and 2(N-1) negative pairs in the batch. The loss function is minimized when we encode the images such that the cosine similarity score $sim(u,v)$ is maximized for u and v that are augmented from the same original image and similarity is minimized for negative pairs coming from different original images.

Focusing on the loss for a positive pair of examples and assume that we have ℓ_2 normalized the representations of the images, the vectorized form of the negative loss function is:

$$u^Tv^+/\tau - log\sum_{v\in v^+,v^-}exp(u^Tv/\tau)$$

Where u is an image vector, v is the set of images in the batch excluding u, v^+ is the one positive pair in v and v^- are the 2(N-1) remaining negative examples.

The loss is reminiscent of softmax loss as well as the loss from word2vec. The loss is an alteration on softmax because just like softmax it uses sum of exponentials to normalize as well as to accentuate the advantage of the one correct positive pair. It's similar to word2vec because it's an unsupervised loss and we do not need labels of training data to compute this loss. Just like how word2vec uses empirical counts of co-occurrences of words to weight different contexts, here NT-Xent uses similarity scores to weight the positive and negative pairs.

In [0]:
```
import torch
import torch.nn.functional as F

# Implmentation in pytorch
class NT_Xent(nn.Module):
  def __init__(self):
    super(NT_Xent,self).__init__()

  def forward(self,o,o_tilde):
    # The inputs o and o_tilde are the representations of images and augmented images
    # passed through encoder

    # normalize feature vectors
    o = F.normalize(o,dim=1)
    o_tilde = F.normalize(o_tilde,dim=1)
    # Compute similarity matrix
    uv = torch.cat([o,o_tilde],dim=0)
    uv_t = torch.t(uv)
    vu = torch.cat([o,o_tilde],dim=0)
    tau = 0.1 # chosen because 0.1 had the best Top1 accuracy according to results from paper.
    sim_mat = torch.exp(torch.div(torch.mm(uv,uv_t),tau))
    # Compute softmax
    sim_sums = torch.sum(sim_mat,dim=1)
    self_sim = torch.diag(sim_mat)
    denoms = sim_sums - self_sim
    nums= torch.exp(torch.div(torch.nn.CosineSimilarity()(uv,vu),tau))
    softmaxes = torch.div(nums,denoms)
    # Compute negative loss
    neglog_losses = -torch.log(softmaxes)
    loss = torch.mean(neglog_losses)
    return loss
```

A simple alternative to this loss function is to use a different similarity measure. I propose to replace the cosine similarity with negative euclidean distance. The only change we need to make to the loss function is the similarity function, whereby we replace it with:

$$sim(u,v) = -\|u-v\|$$

Negative Euclidean and Cosine Similarity captures different aspect of similarities and dissmilarities among the vectors. When vectors are collinear, cosine similarity cannot distinguish the difference among the vectors while euclidean distance can, but cosine similarity can detect the differences in angles which Euclidean distance cannot. In higher dimensions, vectors are likely to be far apart with very large Euclidean distance anyway, so some might argue that cosine similarity is better.

In this case, we can try using negative euclidean distance as the similarity measure to see if it captures the features created by certain simpler affine transformations on the images such as scaling and shifting better than cosine similarity does.

In [0]:
```
import torch
import torch.nn.functional as F

class NT_Xent_euc(nn.Module):
  def __init__(self):
    super(NT_Xent_euc,self).__init__()

  def forward(self,o,o_tilde):
    # The inputs o and o_tilde are the representations of images and augmented images
    # passed through encoder

    # normalize feature vectors
    o = F.normalize(o,dim=1)
    o_tilde = F.normalize(o_tilde,dim=1)

    # Compute similarity matrix
    uv = torch.cat([o,o_tilde],dim=0)
    uv_norm = (uv**2).sum(1).view(-1,1)
    uv_norm_prime = uv_norm.view(1,-1)
    sim_mat = -1 * (uv_norm + uv_norm_prime - 2.0*torch.mm(uv,torch.transpose(uv,0,1)))

    # Compute softmax
    sim_sums = torch.sum(sim_mat,dim=1)
    self_sim = torch.diag(sim_mat)
    denoms = sim_sums - self_sim
    num = -1*(torch.pow(o-o_tilde,2).sum(1))
    nums = torch.cat([num,num],dim=0)
    softmaxes = torch.div(nums,denoms)

    # Compute negative loss
    neglog_losses = -torch.log(softmaxes)
    loss = torch.mean(neglog_losses)
    return loss
```