İlk kısımda bir CNN'den ResNet elde ediyor skip connections'ı tanımlıyor vesaire. Sonuçta Resnet Mnist örneğini yapıyor. DenseNet (Skip connections'a benziyor 2 convolutionda bir inputları toplamak yerine concat ediyor.) ve UNetten bahsediyor.

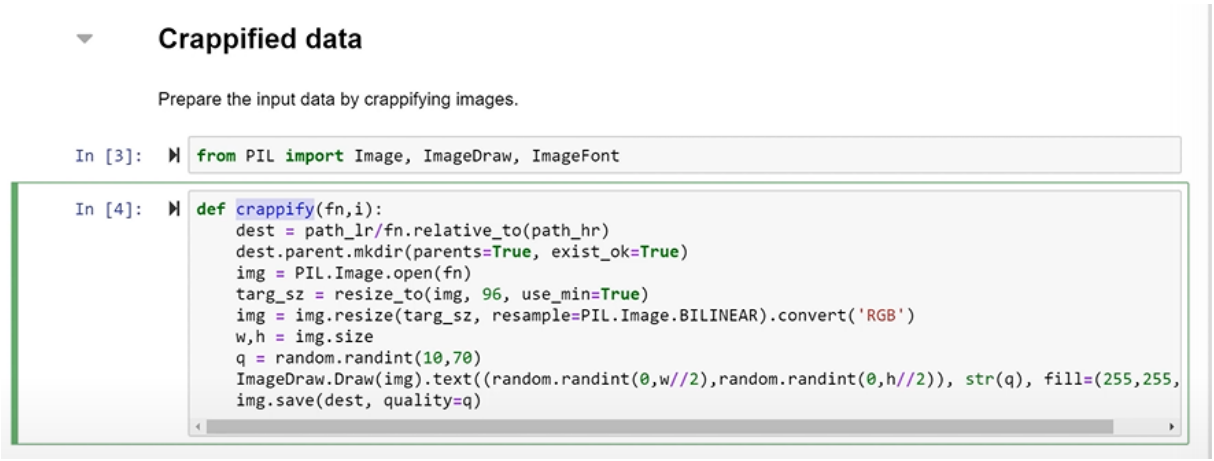
Şimdi ise Image Restoration'a bakacağız:

Image Restoration'da image alınır ancak Segmentation yapılamak yerine yani her pixel classify edilmek yerine daha iyi bir image yaratılır.

Bu bağlamda yapılabilecek çeşitli şeyler var, örneğin low res image alınıp high res yapılabilir, black and white image alınıp coloured yapılabilir, image'dan bazı kısımları kesebiliriz, style transfer yapabiliriz. Bunların hepsi Image to Image Generation tasks, bu kısımdan sonra bunları anlamış olacağız.

Biz Kaggle örneğinde low resolution, üzerinde yazılar olan jpegleri alıp high resolution removed text images'a çevirmek istiyoruz.

Bu örneği yapabikmek için bize shitty images lazım ve her bir image'a karşılık bunların kaliteli high res versiyonlarının label olarak sağlanması lazım. Bunu sağlamanın bir yolu temiz high res images'ı alıp crappfiy etmek kodun ilk kısmında yapılan bu.



Crappified data

Prepare the input data by crappifying images.

In [3]:
```
from PIL import Image, ImageDraw, ImageFont
```

In [4]:
```
def crappify(fn,i):
    dest = path_lr/fn.relative_to(path_hr)
    dest.parent.mkdir(parents=True, exist_ok=True)
    img = PIL.Image.open(fn)
    targ_sz = resize_to(img, 96, use_min=True)
    img = img.resize(targ_sz, resample=PIL.Image.BILINEAR).convert('RGB')
    w,h = img.size
    q = random.randint(10,70)
    ImageDraw.Draw(img).text((random.randint(0,w//2),random.randint(0,h//2)), str(q), fill=(255,255,
    img.save(dest, quality=q)
```

Burada yapılan şu, PIL.Image.opne ile image'ı açıyoruz, 96x96 boyutda küçük bir image olacak şekilde resize ediyoruz. Reszie'ı bilenar interpolation ile yapıyoruz. Daha sonra 10-70 arasında random number seçiyoruz bu number'ı image'ım üzerine random bir location'a çiziyorum. Ve resmi de image'ın jpeg quaility'sinde kaydediyorum ki mesela jpeg quality of 10 is absolute rubbish-70 jpeg quality ise fena değil.

Now let's pretrain the generator.

In [7]:
```
arch = models.resnet34
src = ImageImageList.from_folder(path_lr).random_split_by_pct(0.1, seed=42)
```

In [8]:
```
def get_data(bs,size):
    data = (src.label_from_func(lambda x: path_hr/x.name)
           .transform(get_transforms(max_zoom=2.), size=size, tfm_y=True)
           .databunch(bs=bs).normalize(imagenet_stats, do_y=True))

    data.c = 3
    return data
```

In [9]:
```
data_gen = get_data(bs,size)
```

In [10]:
```
data_gen.show_batch(4)
```

Sonuçta low quality images karşılığında high quality images label olarak bir databunch oluşturduk.

Crappification yavaş gerçekleşebiliri ama FASTAI'ın parallel crappify'i ile crappify fonksiyonunu ve crappify edilecek itemları verirsek, itemlar paralel şekilde crappify edilebilir.

UNUTMA BU UYGULAMADA CRAPPIFY FONKSİYONU DEĞİŞTİRİRSEK UYGULAMANIN AMACINI DA DEĞİŞTİRMİŞ OLURUZ, EĞER COLORIZATIN YAPMAK İSTİYORSAK CRAPPIFY İÇİNDE IMAGES'I B&W'A ÇEVİREBİLİRİZ, EĞER IMAGE'IN BİR KISIMI HALLUCINATING IMAGES İLE DEGISTIRMEK ISTERSEK, RESMIN BİR KISMINI BUYUK BİR BLACK BOX EKLERİZ, YANİ BİR RESİMDEN DİĞERİNE ELDE ETMEK İSTİYORSAK, CRAPPIFY İÇİNDE BUNU TANIMLAMALIYIZ VE BIR DATABUNCH OLUŞTURMALIYIZ.

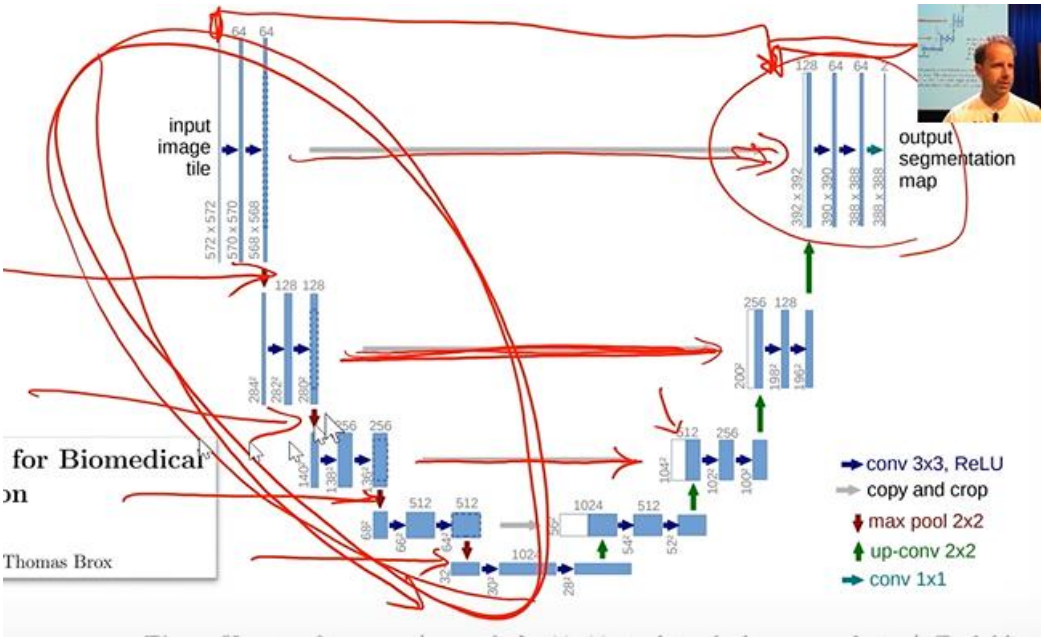
Şimdi yapmak istediğimiz yukarıdaki soldaki gibi crappified images'ı alıp sağdaki gibi daha güzel images veren bir model eğitmek istiyoruz, bunun için Unet kullanmak istiyoruz, Unet segmentation için de kullanılıyordu avantajı farklı layerlar arasında skip connections sağlamasıydı, yani son layerda orijinal image'a da bakarak bir generation yapmasını bekliyoruz.

```
In [11]: wd = 1e-3

In [12]: y_range = (-3.,3.)

In [13]: loss_gen = MSELossFlat()

In [14]: def create_gen_learner():
             return unet_learner(data_gen, arch, wd=wd, blur=True, norm_type=NormType.Weight,
                                 self_attention=True, y_range=y_range, loss_func=loss_gen)

In [15]: learn_gen = create_gen_learner()
```

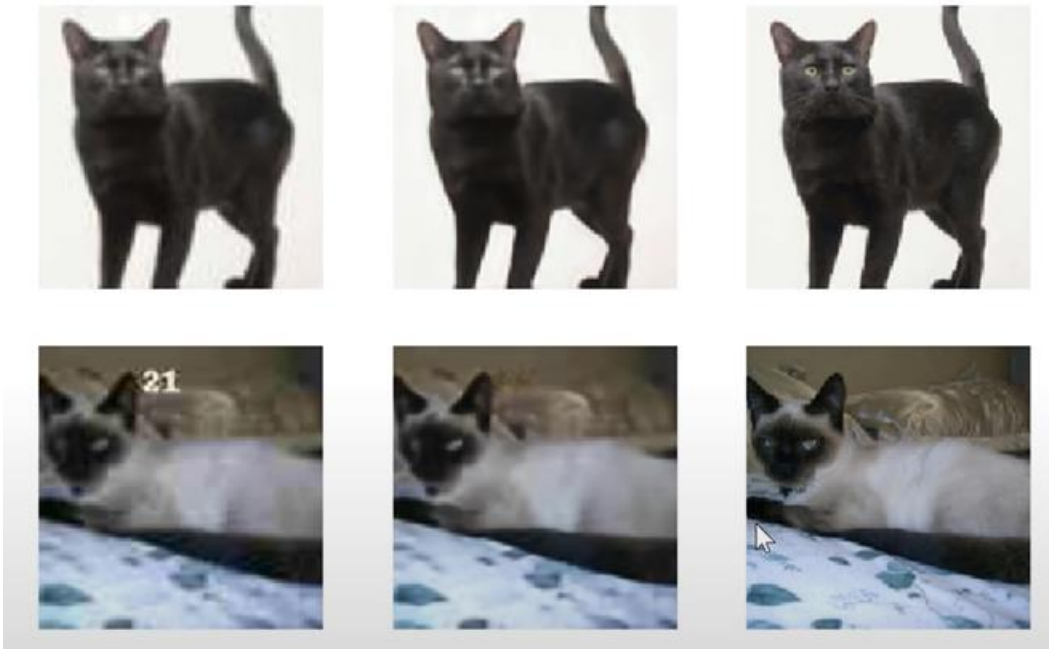
Architecture olarak pretrained resnet34 kullanıyoruz. Segmentation'da her pixel için classification yapılıyordu burda ise her pixel için MSE error hesaplanıyor.

MSELossFlat kullanıyoruz çünkü MSE normalde 2 vektör bekler 2 images'a baktığımız için önce flatten ediyoruz.

Default olarak unet'i yaratınca pretrained part freeze edilir, yani fit_one_cycle dediğimizde resnet34 eğitilmez yeni eklenen layerlar eğitilir, Unet için yeni eklenen layerlar ise eskisi gibi 2 tane FC layer değil aşağıdaki resimdeki U'nun sağ kısmı:



Modeli yarattıktan sonra önce freezed şekilde bir eğitim yapıyoruz sonra unfreeze ile fine-tuning yapıyoruz yani aynen eskisi gibi sonuçta gayet iyi bir model elde ediyoruz.

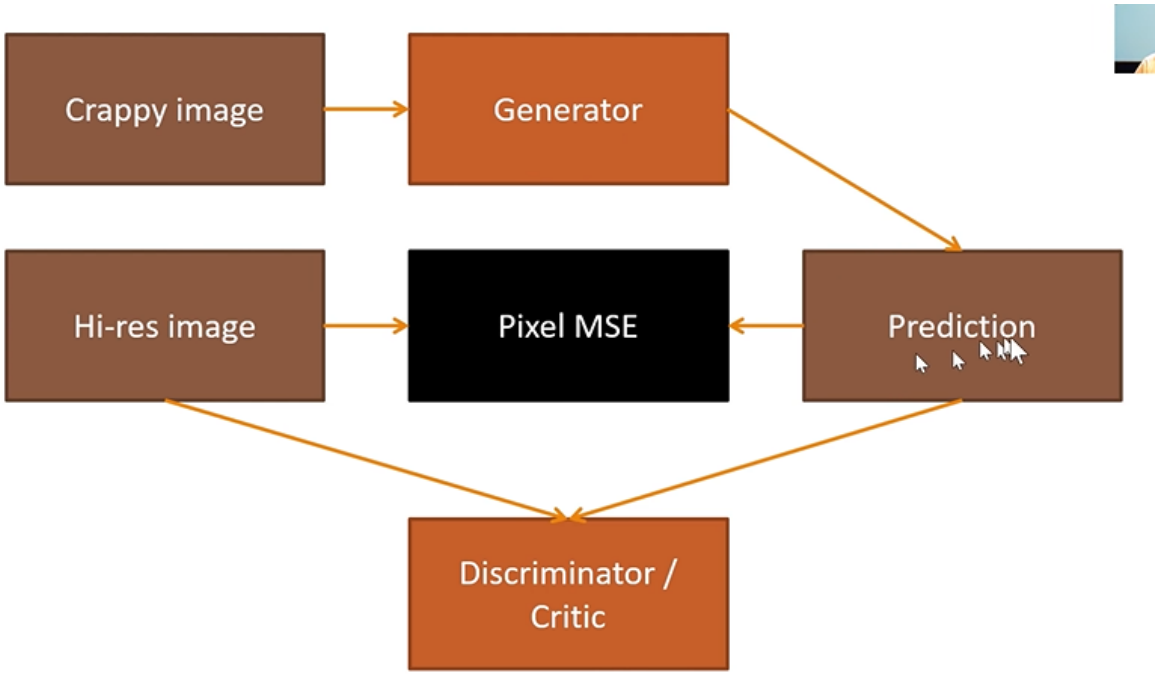


Model watermark'ı kaldırmakta başarılı ama resolution'ı pek de arttırdığını söyleyemeyiz, label ile 2. Resim arasında hala büyük bir resolution farkı var. Bunun sebebi loss function'ın aslında tam olarak bizim istediğimiz şeyi tanımlamıyor oluşu, pixeller arasında MSE aslında şuan gayet küçük, her pixelle baktığımızda bu rakam gayet iyi ama yine de detayları kaybediyoruz, bizim istediğimiz loss function image'ın good quality picture olup olmadığına bir şekilde karar veremeli ki ona göre loss atayabilsin.

GAN bu problemi çözmek için başka bir Model'i çağıran bir loss function'dan yararlanır yani GAN modelde 2 model çalışır, bir tanesi yukarıdaki gibi bir image üretirken diğeri üretilen fotoğrafın lossunu hesaplamak için kullanılır.

Bunu daha iyi anlayalım:

Elimizde crappy images ve generator'ımız var, generator'ımız çok iyi çalışmıyor ama fena değil. Ayrıca label olarak elimizde Hi-res image'ımız da var, Pixel MSE ile hi-res image ile predictionu karşılaştırabiliyoruz. Ayrıca Discriminator veya Critic isminde yeni bir model eğitebiliriz bunun yapacağı şey de hi-res image'ı ve Prediction'ı alıp binary classification yapmak yani hangisinin generated olduğunu anlamak. Bildiğimiz regular binary cross entropy loss ile binary classification yapacak.



Böylece generator'ımız loss olarak critiğin confidence'ını kullanacak critiğin confidence'ı ne kadar yüksek ve doğruyu bildiyse generator'ımıza o kadar yüksek loss gitmeli ki kendini daha iyi images yaratmaya zorlasın.

Burada izlenecek plan şu önce generator'ı biraz eğiteceğiz ve bir süre sonra critiği kolayca kandıracak hale gelecek daha sonra critiği eğiteceğiz böylece daha zor kandırılacak yani iki model de sürekli birbirini daha iyi hale getirecek.

Generator'ımız ve Discirimator'ımız pretrained, bu daha önce yapılmamış bir şey, zaten GAN'ı eğitmek çok zodur sebebi de başlangıç eğitiminin çok zor olması çünkü körler sağırlar birbirini ağırlar oluyor. Ancak pretrained olunca ikisi de birbirini daha iyiye taşımak için biçilmiş kaftan.

Şimdi bir critic modeli oluşturalım bu bildiğimiz standard fastai binary classification model'den fazlası değil, iki foldera ihtiyacımız var biri generated images'dan oluşmalı diğeri high res images'dan, high-res images zaten cepte, dolayısıyla generated images'ı da bir klasöre kaydettiğimiz zaman critic dataseimiz hazır olacak.

Save generated images

In [21]:
```
learn_gen.load('gen-pre2');
```

In [22]:
```
name_gen = 'image_gen'
path_gen = path/name_gen
```

In [23]:
```
# shutil.rmtree(path_gen)
```

In [24]:
```
path_gen.mkdir(exist_ok=True)
```

In [25]:
```
def save_preds(dl):
    i=0
    names = dl.dataset.items

    for b in dl:
        preds = learn_gen.pred_batch(batch=b, reconstruct=True)
        for o in preds:
            o.save(path_gen/names[i].name)
            i += 1
```

In [26]:
```
save_preds(data_gen.fix_dl)
```

Kaydedilen generated images'dan birine bakabilirim:

In [27]:
```
PIL.Image.open(path_gen.ls()[0])
```

Out[27]:

Şimdi critiği train etmek için datasetim hazır olduğuna göre training'e başlayabilirim:

Train critic

In [28]:
```
learn_gen=None
gc.collect()
```

Out[28]: 3755

Pretrain the critic on crappy vs not crappy.

In [29]:
```
def get_crit_data(classes, bs, size):
    src = ImageItemList.from_folder(path, include=classes).random_split_by_pct(0.1, seed=42)
    ll = src.label_from_folder(classes=classes)
    data = (ll.transform(get_transforms(max_zoom=2.), size=size)
            .databunch(bs=bs).normalize(imagenet_stats))
    data.c = 3
    return data
```

In [30]:
```
data_crit = get_crit_data([name_gen, 'images'], bs=bs, size=size)
```

Jupyter notebook'da in order to reclaim GPU memory we need to restart the jupyter notebook, bu annoying yukarıdaki ilk iki satır bunun için yapılmış. Çok GPU kullandığını bildiğimiz learn_gen'i None olarak atarsak ve gc.collect() dersek python memory garbage collection yapar ve GPU'yu rahatlatmış oluruz.

Standard bir object classifier yaratıyoruz, farklı bir şey yok bunun için önce dataset'i yarattık show batch ile bakabiliriz:

```
In [31]: data_crit.show_batch(rows=3, ds_type=DatasetType.Train, imgsize=3)
```

Loss function'ı tanımlıyoruz, ancak burada architecture olarak resnet kullanmayacağız, detaylarına part2'de gireceğiz diyor. Fastai'ın sağladı gan_critic() architecture'ı kullanıyoruz. Gan'ı kullanırken loss'u adaptiveloss ile Wrap etmeliyiz bunun da detayına part2'de gireceğiz.

```
In [32]: loss_critic = AdaptiveLoss(nn.BCEWithLogitsLoss())

In [33]: def create_critic_learner(data, metrics):
             return Learner(data, gan_critic(), metrics=metrics, loss_func=loss_critic, wd=wd)

In [34]: learn_critic = create_critic_learner(data_crit, accuracy_thresh_expand)

In [35]: learn_critic.fit_one_cycle(6, 1e-3)
```

Total time: 09:40

epoch	train_loss	valid_loss	accuracy_thresh_expand
1	0.678256	0.687312	0.531083
2	0.434768	0.366180	0.851823
3	0.186435	0.128874	0.955214
4	0.120681	0.072901	0.980228
5	0.099568	0.107304	0.962564
6	0.071958	0.078094	0.976239

Metric'de gan'a özel şekilde biraz farklı tanımlandı ama mantık aynı. Daha sonra eğitim yapıyoruz, görüyoruz ki critiğimiz %97.6 ile doğru ayrım yapıyor.

Artık elimizde bir pretrained Generator ve pretrained Critic var

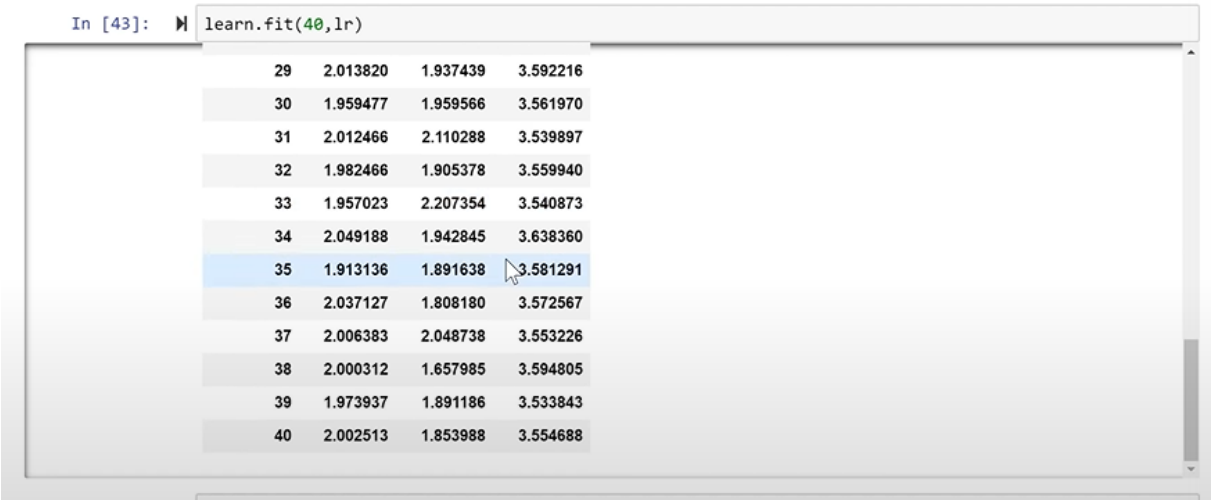
Bu noktadan sonra amacımız bir birini bir diğerine eğiterek ikisininde giderek daha iyi olmasını sağlamak:

Hangi modeli ne kadar eğiteceğiz, nasıl bir learning rate vereceğiz bunlar hala muğlak bu yüzden fastai bir GANLearner tanımlamış basitçe generator'ımızı ve critiğimizi buraya pass ediyoruz, sonuçta learn.fit dediğimizde hangisini ne kadar eğiticeğini ne zaman diğerine döneceğini vesaire kendi ayarlıyor.

Burada önemli bir nokta loss function olarak sadece critiği kullanmıyoruz eğer böyle yaparsak generator critiği kandırmak için gerçekçi fotoğraflar üretebilir ancak bunların orjinalleri ile alakası olmayabilir, yani Generator'ı eğitirken Pixel MSE'nin de critic loss'la birlikte bu süreçte kullanılması lazım.

GAN kullanıyorska yukardaki hyperparameters'ı kullan diyor.

Gan eğitmenin bir diğer tough noktası şu learn.fit dedik eğitim yapıyoruz, ancak epoch'a karşılık görünen train_loss – gen_loss – disc_loss sayıları bir şey ifade etmez, aşağı yukarı sabit kalmaları beklenen şeydir çünkü iki modelde sürekli kendini geliştirir, yani iki model de sürekli zorluğu artan bir dataset üzerinde çalışırlar.



Sonuçta elde edilen images'da bariz bir sharpening ve iyileşme var ancak, modelimiz bu örnekte kedinin gözlerini getirmemiş çünkü bunun önemli bir feature olduğunun farkında değil, onun için ikisi de gerçek kediler, ancak bilmiyor ki biz kedinin gözlerini görmek istiyoruz, bunu önemsiyoruz.

Peki bu gözleri nasıl getiririz?

GAN konusunu bitirmek için son notebook'a da bakalım lesson7 WGAN

Bu örnekte sıfırdan yatak odası fotoğrafları oluşturuyoruz input olarak modele bir noise fotoğrafı veriliyor, generator bu noise'dan critiğin gerçeğinden ayırt edemeyeceği bir yatak odası fotoğrafı yaratmaya çalışıyor.

Burada pretraining falan yok, bu biraz old style approach. Birkaç saat eğitimden sonra badroom'a benzeyen images üretiliyor. Bu notebook'a da bakabilirsin diyor.

Fastai ile hızlıca ve robust GAN yapıları üretebiliyoruz bunu anladık ve bu iyi ancak Jeremy bir yandan da bundan tatmin olmadığını söylüyor çünkü training'i çok uzun sürüyor ve outputs da o kadar da iyi değil.

Bu yüzden hiç GAN kullanmadan, GAN'ın yaptığı resolution artıtrma vb gibi işleri yapabilsek iyi olurdu, Feature Losses kavramından yararlanarak bunu yapabiliyoruz.

Perceptual Losses for Real-Time Style Transfer and Super-Resolution

Justin Johnson, Alexandre Alahi, Li Fei-Fei

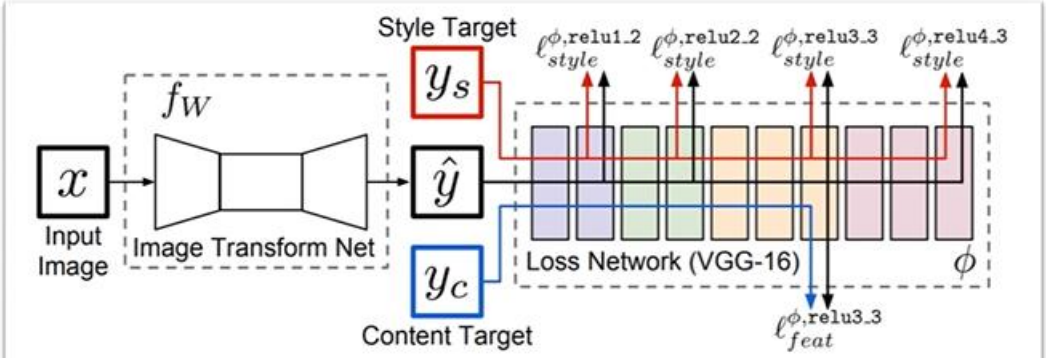


Fig. 2. System overview. We train an *image transformation network* to transform input images into output images. We use a *loss network* pretrained for image classification to define *perceptual loss functions* that measure perceptual differences in content and style between images. The loss network remains fixed during the training process.

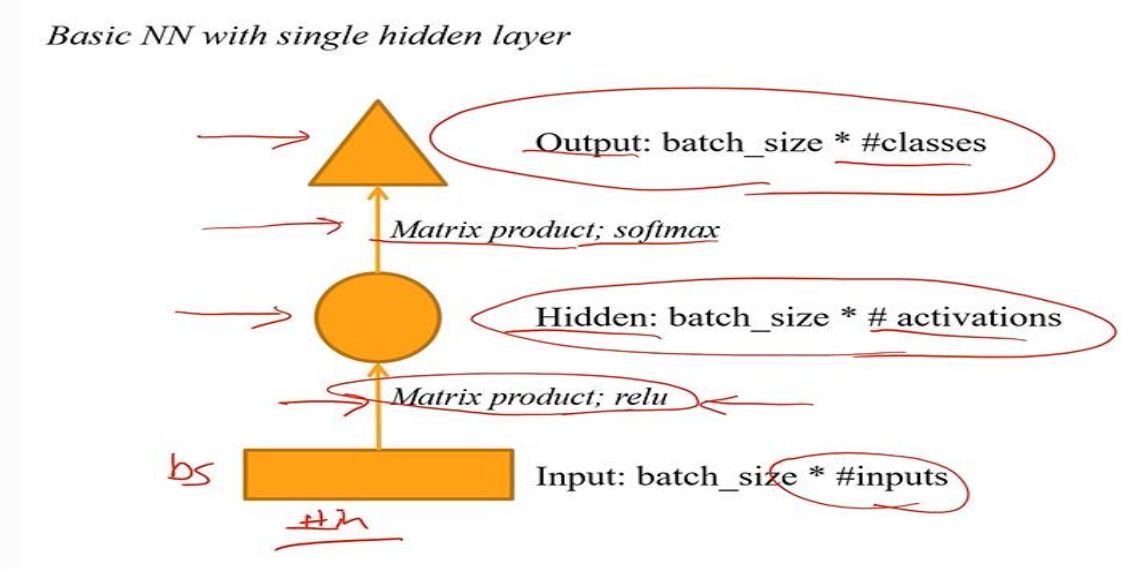
Burada yapılan şey şu daha önce olduğu gibi input'u decoder ve encoder'dan oluşan bir generator'a vermek. Buradaki yenilik loss function'ı tanımlamakta loss function'ı tanımlarken kedi gözü gibi eksik feature'lara ceza kesebilirsek generatorumuz daha iyi iş çıkaracaktır işte bu amaçla yukarıda görüldüğü gibi bir başka pretrained model VGG16 kullandık aslında bu eski sayılır ama iş görüyor. Bu pretrained network'ün ara aktivasyonlarını hem generated image hem de target için alıp karşılaştırırsam buna göre bir loss çıkarabilirim yani yapılan iki image'ın deep feature'larını karşılaştırmak, bunun yanında pixel MSE de kullandığımız zaman gayet iyi sonuçlar elde ediyoruz.

Hatta bu yaklaşımı GAN yaklaşımı ile birleştirirsek daha da iyi sonuçlar elde edebiliriz.

Bu örnek için Lesson7 SuperRes örneğini kullanıyoruz.

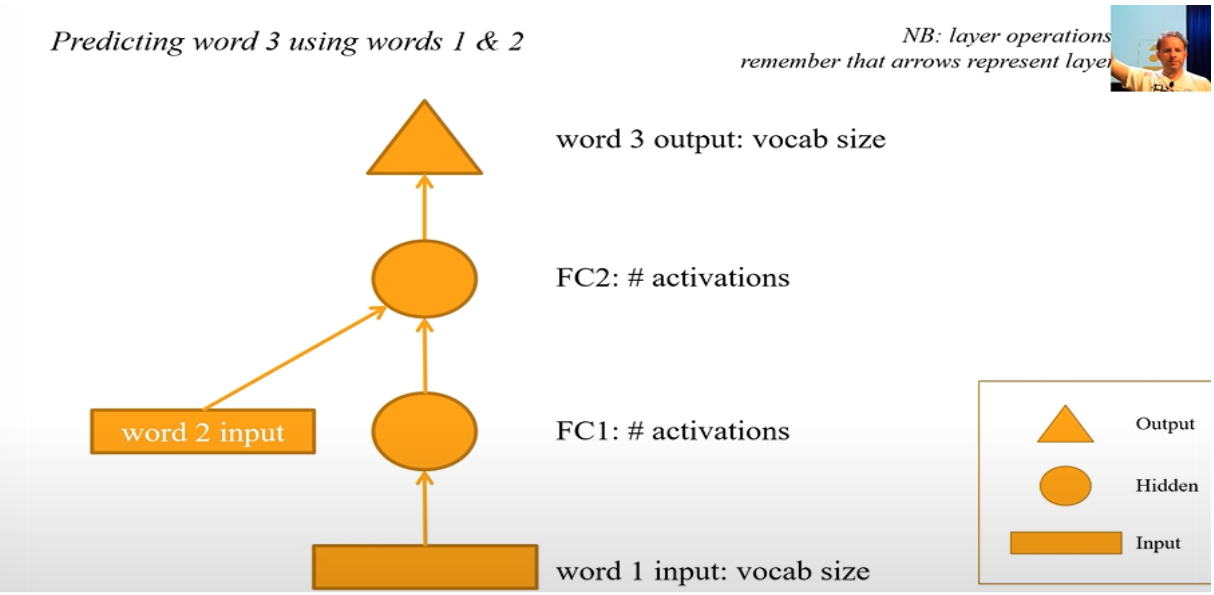
Bu örnekte GAN kullanmadan kötü fotoğrafları iyileştirmeye çalışıyor, ve mükemmel sonuçlar elde etti bunu yapmak için style transfer'de gördüğüm feature loss kavramını devreye soktu yani loss'u hesaplamak için generator'ın ürettiği image pixel loss'un yanında bir de pre-trained bir modelin içine veriliyor ve aradaki feature'lar, target'in feature'ları ile karşılaştırılıyor, böylece model feature kayıplarını loss function'a yansıtabiliyor ve generator'ımız bir critiğe ihtiyaç duymadan iyi bir loss functionla kendini bariz bir şekilde iyi hale getirebiliyor.

Son olarak RNN’s den Bahseceğiz:



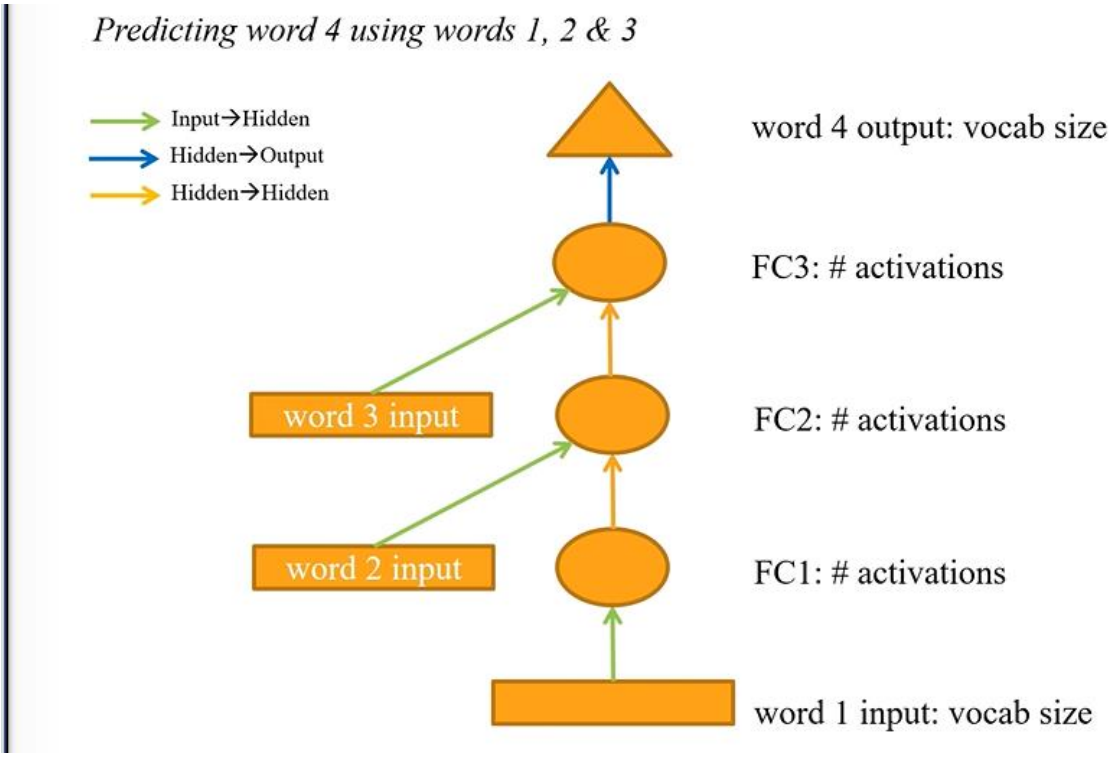
- Dikdörtgen input’u temsil edecek, batch size * number of inputs.
- Oklar layerları temsil edecek
- Circle activations’ı temsil edecek.
- Üçgen de outputs’u temsil edecek.

Şimdi bu gösterimi kullanarak şöyle bir model yaratmaya çalıştığımızı varsayalım, çok büyük bir text documanını alıyoruz, her iki kelimeyi input olarak alıyoruz ve sıradaki 3. Kelimeyi de ilgili 2 kelimenin label’ı olarak alıyoruz yani modelin amacı 2 kelime verilince 3.’yü tahmin etmek:



Word1'ı input olarak alıyoruz, bir embedding layer'dan geçiriyoruz, aktivasyonu elde ediyoruz, sonra bu aktivasyonu bir nonlinearity içeren bir layer'dan geçiriyoruz, bu sırada word2'yi al embedding layer'dan geçir daha sonra bu iki aktivasyonu topla veya concatanate et sonuçta fc2 aktivasyonunu elde ediyoruz. Bu aktivasyonu da softmax layer'a beslersek sonuçta word3 outputunu elde edebiliriz.

Bunu bir adım ileriye götürelim peki ya 3 input word ile 4. Kelimeyi tahmin etmek isteseydik? O zaman da benzer bir şekilde aşağıdaki gibi bir NN yapısıyla bunu yapabilirdik.



Jeremy diyor ki yukarıda farklı işlevi yapan layerlar farklı arrow'larla gösterilmiş diyor ki bu aynı renkle gösterilen layerların hepsi aynı weight matrix'i kullanabilir. Çünkü aynı işi yapıyorlar, her bir embedding için farklı bir weight matrix'e gerek yok diyor.

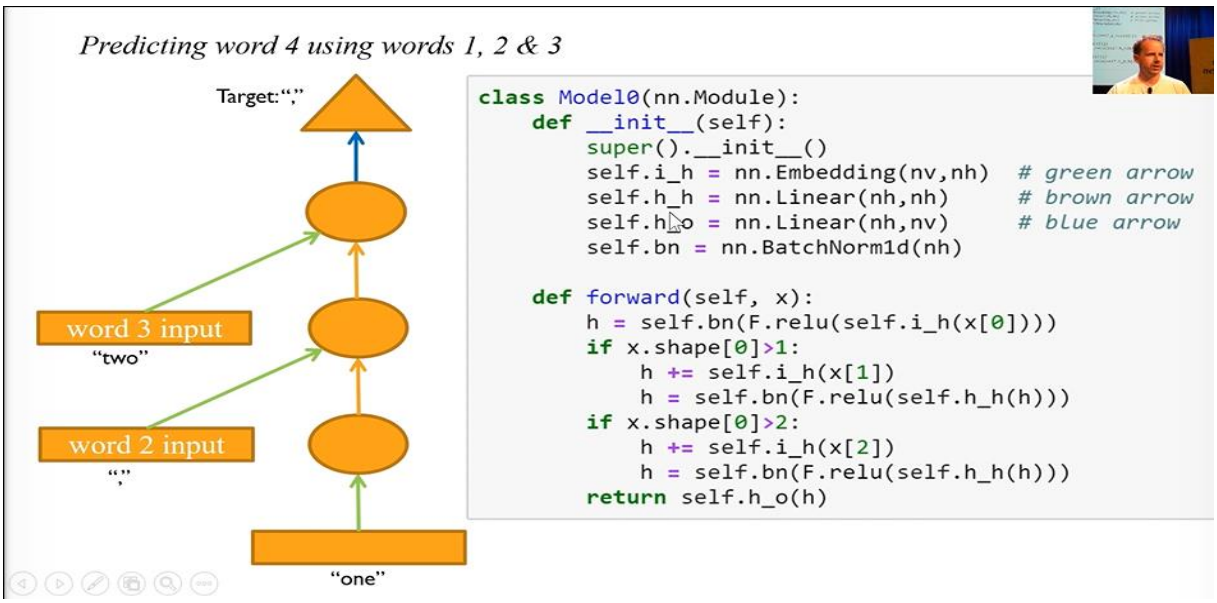
İşte RNN yapısı böyle ortaya çıkıyor, şimdi bu yapıyı oluşturmak için

Lesson 7 Human Numbers kerneline atlıyor.

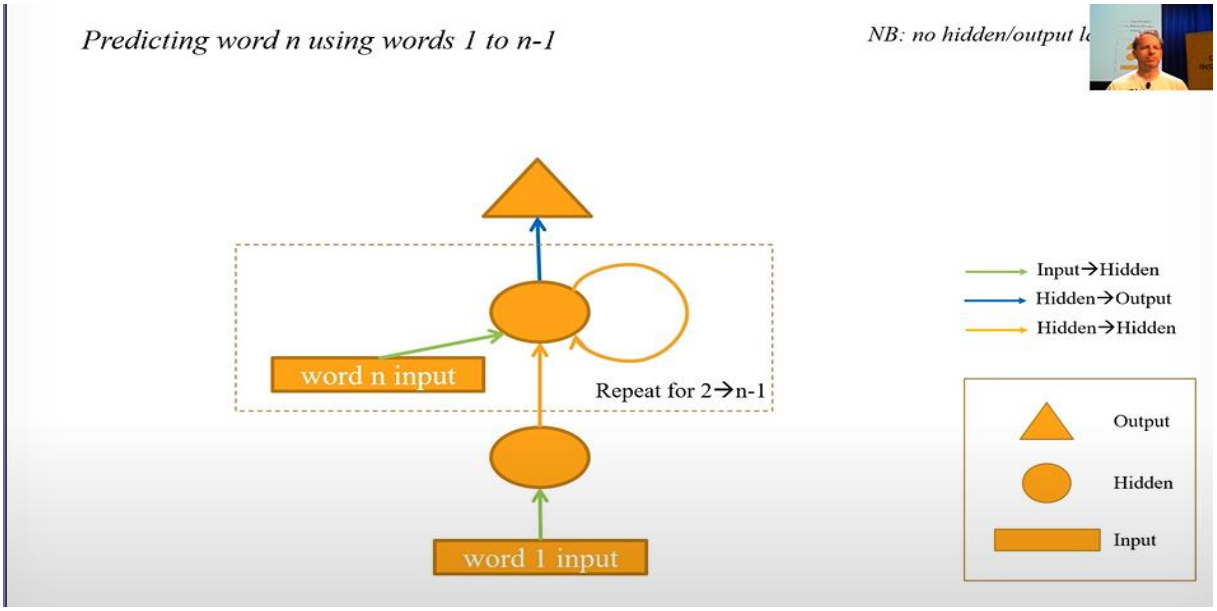
Lesson 7 Human Numbers

Bu dataset 1'den 9999'a kadar tüm sayıları text olarak içeren bir dataset. Bu dataset üzerinde bir language model eğiterek sıradaki kelimeyi tahmin eden bir model geliştirmek istiyoruz.

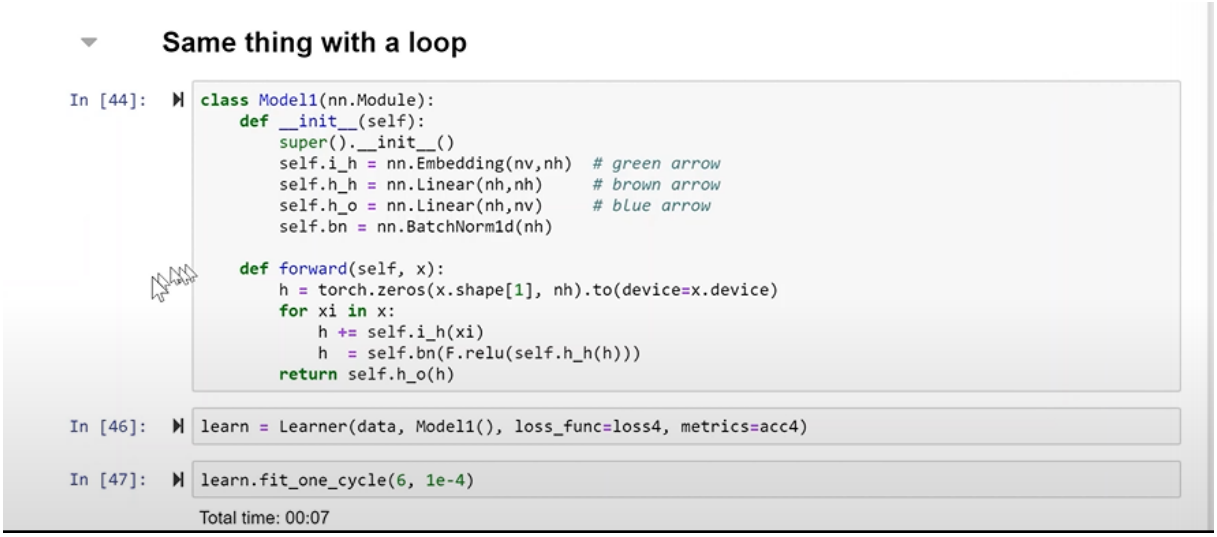
Bu modeli detaylarıyla açıklıyor, gerekince tekrar izle. Şuan çok detayına takılmıyorum.



Bu koddaki tekrar eden şeyler var bundan kurtulmak için refactoring yaparsak ve bir loop kullanırsak elde ettiğimiz şey bir RNN oluyor, yani RNN yukarıdaki yapının refactor edilmiş halinden fazlası değil, yukarıdaki yapı aşağıdakine dönüşüyor:



Aşağıdaki versiyonda kodun refactor edilmiş hali yani RNN koduna dönüşmüş oldu.



Buraları RNN ile işin olursa tekrar detaylıca izleyebilirsin ben şuan sadece bir giriş olsun diye bakıyorum.