APS360 Fundamentals of AI

Lisa Zhang

Lecture 8; June 3, 2019

Agenda

Last time:

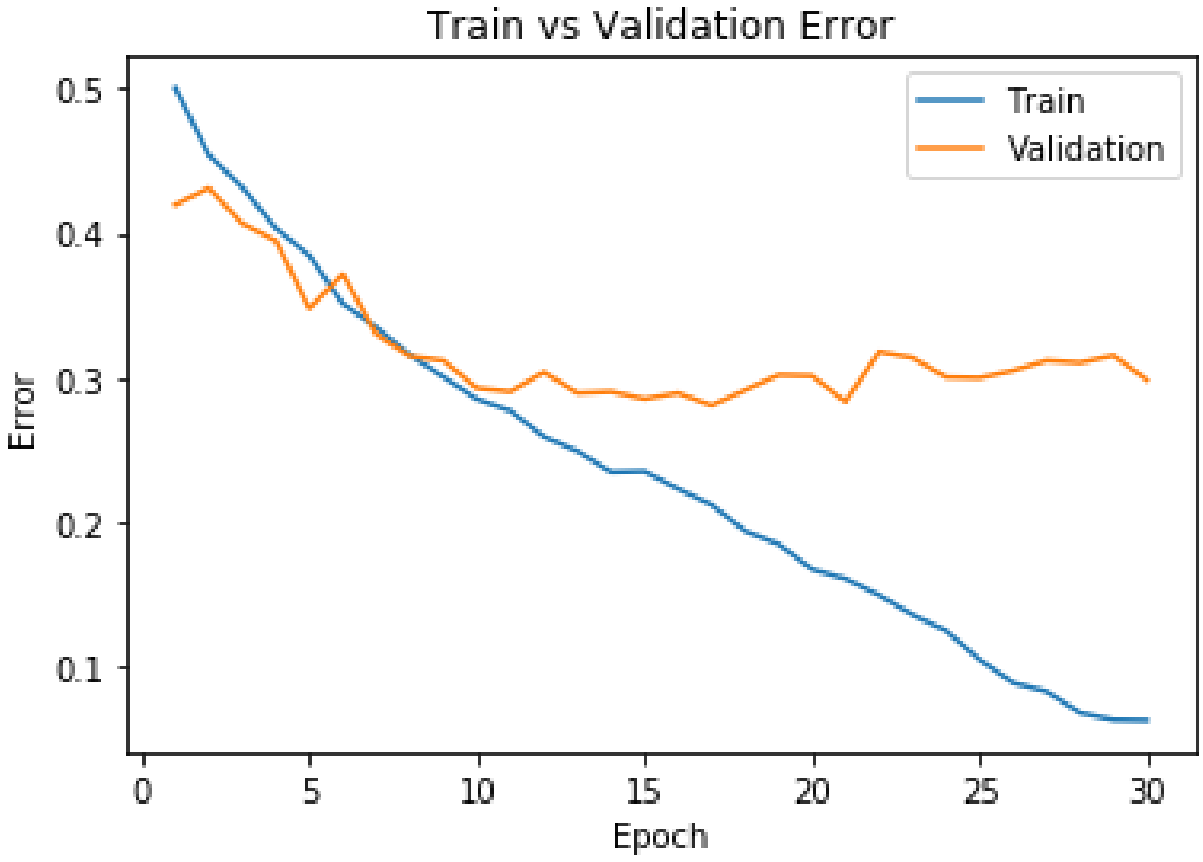
- Convolutional Architectures
- Discussions about Lab 3

Today:

- First Hour: Preventing Overfitting
- Second hour:
  - Transpose Convolutions
  - Autoencoders – first unsupervised learning, generative model!

Preventing Overfitting

Overfitting



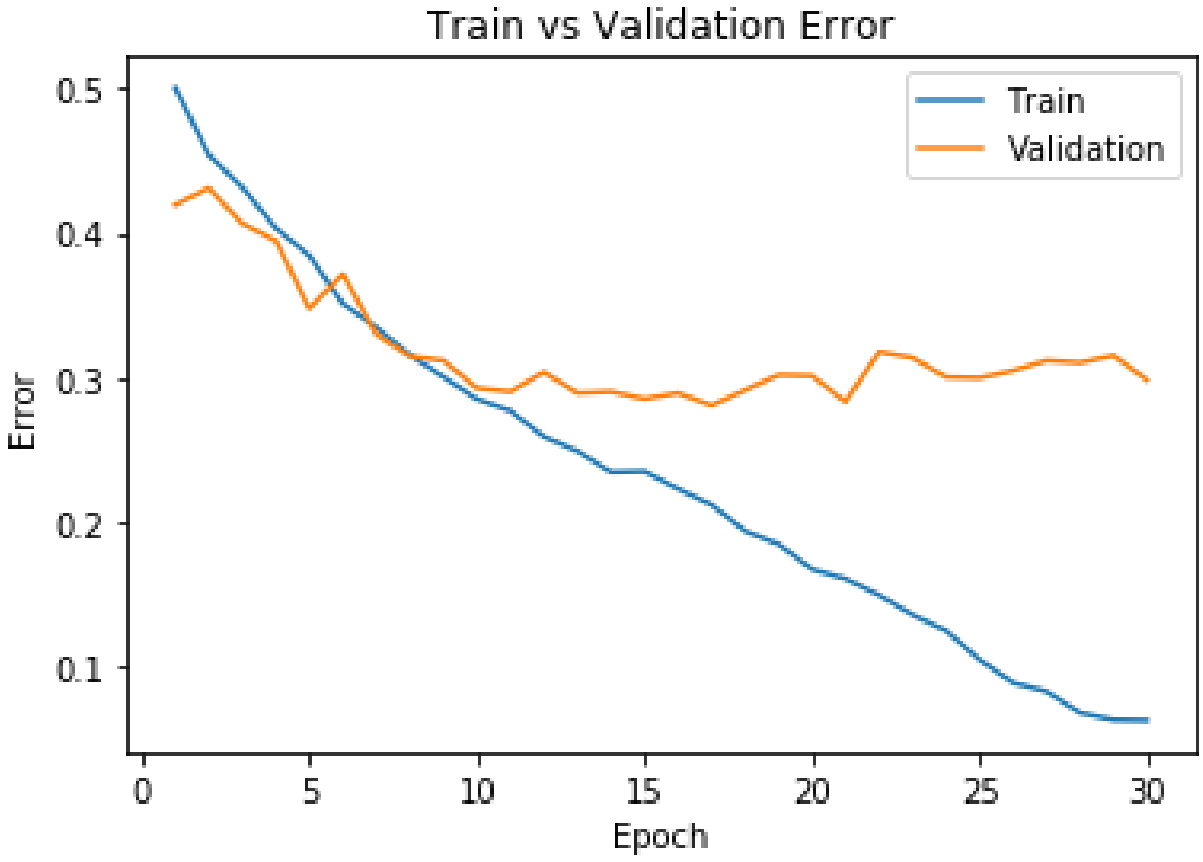
Neural network learns too much about the idiosyncracies in the training set that does not generalize.

Q: What factors can increase the chance of overfitting?

Q: What factors can increase the chance of overfitting?

- fewer weights or more weights?
- fewer layers or more layers?
- less training data or more training data?
- fewer training iterations or more training iterations?
- fewer artificial neurons or more artificial neurons?

Underfitting



Neural network does not capture the underlying trend of the data – there are more generalizable patterns that can be learned.

Detecting Underfitting

- Detecting underfitting is harder than detecting overfitting
- Typically, people choose a model that will overfit, then apply techniques to reduce overfitting

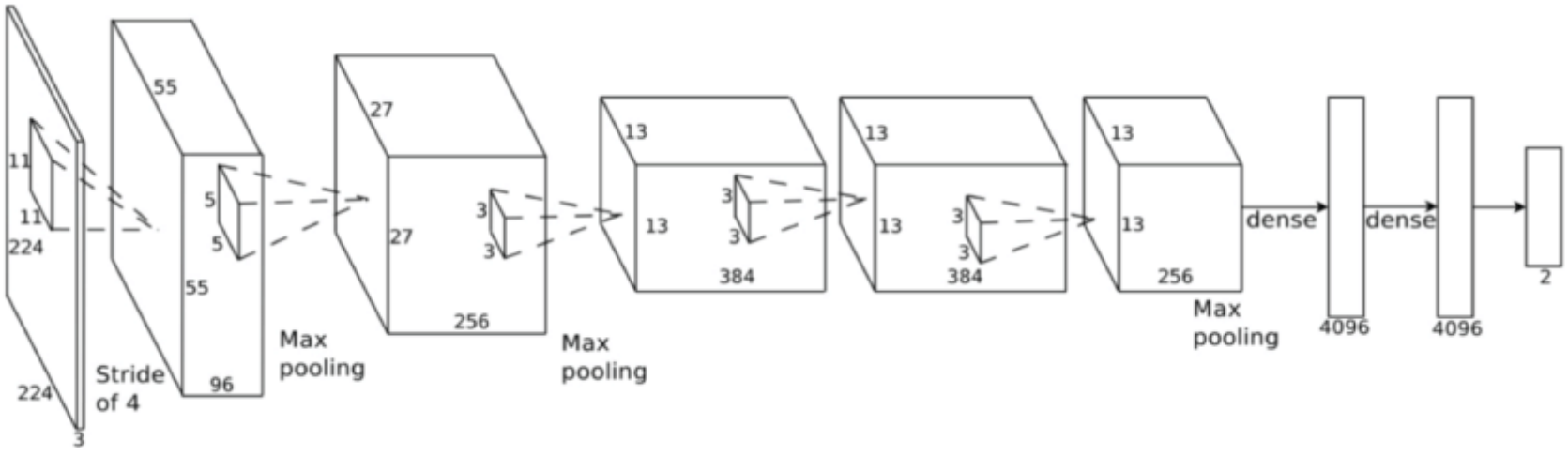
Preventing Overfitting

Q: What strategies have we discussed for preventing overfitting?

Ideas to Prevent Overfitting

- Use a larger training set (expensive, often not feasible)
- Use a smaller network (requires starting over, might underfit)
- Weight-sharing - as in convolutional neural networks
- **Early stopping** - stop training at an earlier epoch
- **Transfer learning** - a network that is pre-trained on a larger data set

Transfer Learning Features



- Each layer we computes a different *representation* of the input
- These *representations* are better-suited (to the classification task) than the input representation
- These *representations* turns out to be useful to other tasks!

Other Strategies

- Data Normalization (indirectly, by solving an easier problem)
- Data Augmentation
- Weight Decay
- Model Averaging
- Dropout

Data Normalization

Normalization: adjust input feature values to be scaled similarily

- All the input features should have similar means and standard deviations
- Less of an issue for images (since all features are pixels), but could be an issue for other types of data:
  - e.g. predicting housing sales price based on number of bedroom, square footage, ...
  - the input feature should be scaled similarly

For images, having each pixel in the range `[0, 1]` is usually fine.

Normalization Transforms

In Lab 2 we used this transformation:

```
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
```

- subtracts each pixel by 0.5 and divides the result by 0.5.
- each pixel would be in the range `[-1, 1]`
- we have both positive and negative features

You're welcome to use this normalization for lab 3.

Data Augmentation

Make small alterations to the training data to obtain "new data":

- Flip each image horizontally or vertically (e.g. for cats vs dogs, not for gesture recognition)
- Shift each pixel a little to the left or right
- Rotate the images a little
- Add noise to the image
- Combination of the above

Data Augmentation for Digit Recognition

- See code

Penalizing Large Weights

Penalize **large weights**, by adding a term (e.g. $\sum_k w_k^2$) to the loss function

Q: Why is it not ideal to have large (absolute value) weights?

Penalizing Large Weights

Penalize **large weights**, by adding a term (e.g. $\sum_k w_k^2$) to the loss function

Q: Why is it not ideal to have large (absolute value) weights?

Because large weights mean that the prediction relies **a lot** on the content of one pixel (or one feature)

Weight Decay

- L^1 regularization: add a term $\sum_k |w_k|$ to the loss function
  - Mathematically, this term encourages weights to be exactly 0
- L^2 regularization: add a term $\sum_k w_k^2$ to the loss function
  - Mathematically, in each iteration the weight is pushed towards 0
- Combination of L^1 and L^2 regularization: add a term $\sum_k |w_k| + w_k^2$ to the loss function

Weight Decay in PyTorch

L^2 weight decay is usually a setting of the optimizer

```
optim.SGD(model.parameters(), weight_decay=0.001, ...)
```

See code

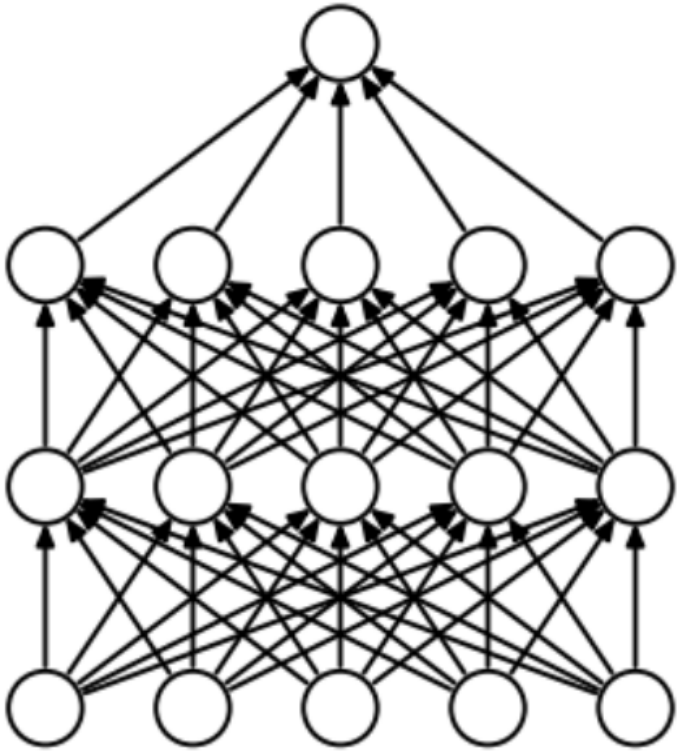
Model Averaging

To prevent overfitting, build **many** models, and average their predictions.

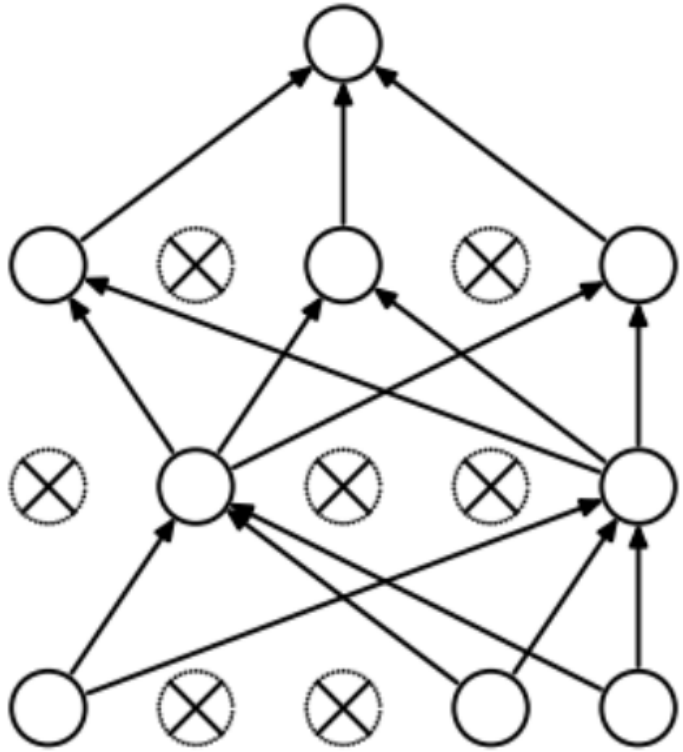
Each model use a slightly different architecture, different initial weights, or different subset of the training data.

Dropout

Randomly "remove" a portion of neurons from each training iteration:



(a) Standard Neural Net



(b) After applying dropout.

A different set of neurons are "removed" in a different iteration.

All neurons are used during test time (for evaluation and for making actual predictions)

Why dropout

- Prevent weights from depending on each other.
- Encourage each hidden unit to learn "more independent" features.
- Is actually a form of model averaging: averaging over all possible connections.

Dropout for Digit Recognition

See code.

Important to set `model.train()` or `model.eval()` to change the behaviour of the dropout layer.

Transpose Convolution

Pixel-wise prediction

A prediction problem where we label the content of each pixel is known as a **pixel-wise prediction problem**

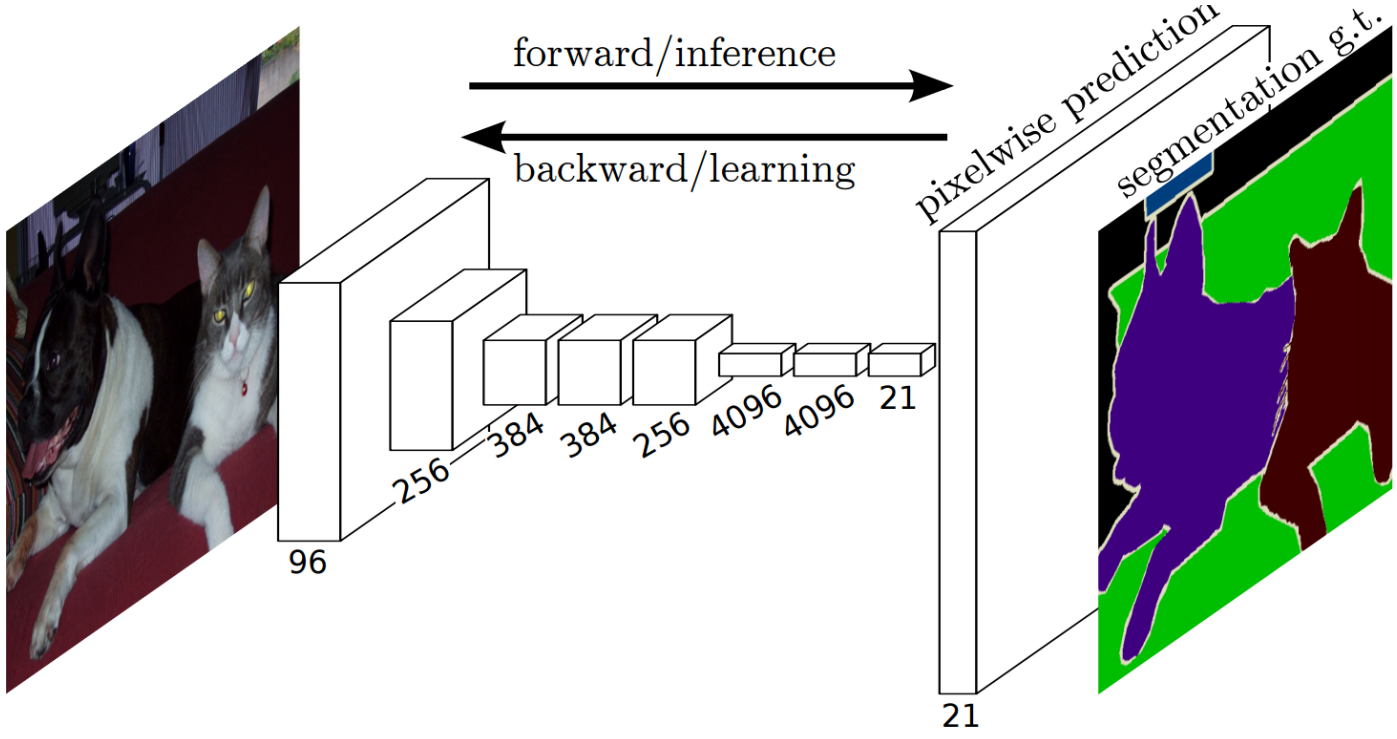


Figure 1: http://deeplearning.net/tutorial/fcn_2D_segm.html

Q: How do we generate pixel-wise predictions?

What we need:

We need to be able to **up-sample** features, i.e. to obtain high-resolution features from low-resolution features

- Opposite of max-pooling OR
- Opposite of a strided convolution

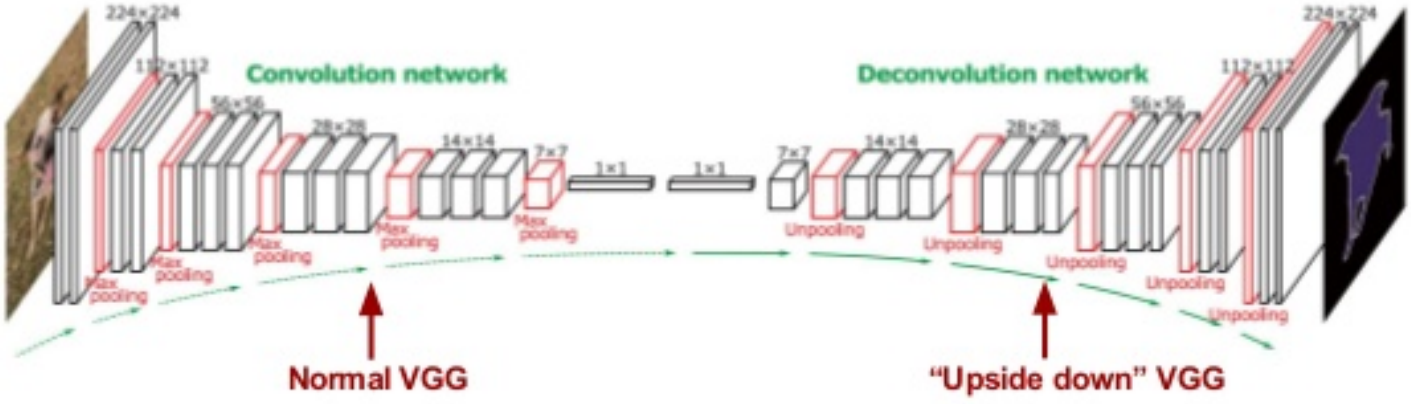
We need an **inverse** convolution – a.k.a a **deconvolution** or **transpose convolution**.

Architectures with Transpose Convolution

More than one upsampling layer

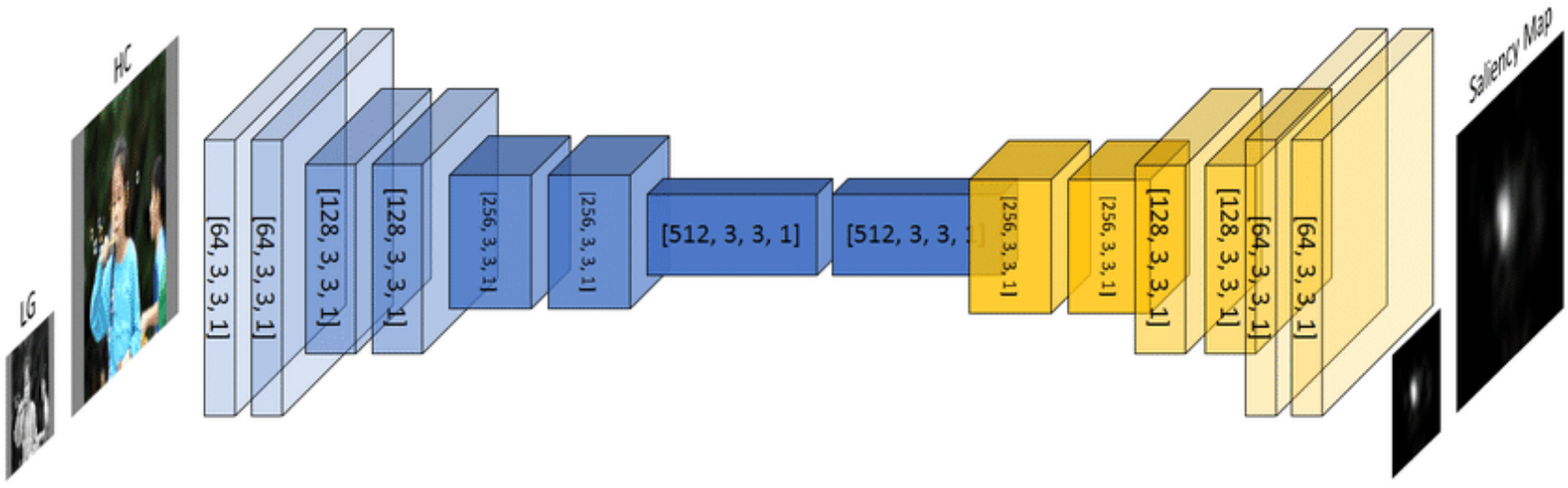
DeconvNet:

VGG-16 (conv+Relu+MaxPool) + mirrored VGG (Unpooling+'deconv'+Relu)



Noh et al, "Learning Deconvolution Network for Semantic Segmentation", ICCV 2015

Architectures with Transpose Convolution 2



Inverse Convolution

```
>>> x = torch.randn(2, 8, 64, 64)
>>> conv = nn.Conv2d(in_channels=8,
...                  out_channels=8,
...                  kernel_size=5)
>>> y = conv(x)
>>> y.shape
```

Inverse Convolution

```
>>> x = torch.randn(2, 8, 64, 64)
>>> conv = nn.Conv2d(in_channels=8,
...                  out_channels=8,
...                  kernel_size=5)
>>> y = conv(x)
>>> y.shape

>>> convt = nn.ConvTranspose2d(in_channels=8,
...                            out_channels=8,
...                            kernel_size=5)
>>> x = convt(y)
>>> x.shape
```

Inverse Convolution

```
>>> x = torch.randn(2, 8, 64, 64)
>>> conv = nn.Conv2d(in_channels=8,
...                  out_channels=8,
...                  kernel_size=5)
>>> y = conv(x)
>>> y.shape

>>> convt = nn.ConvTranspose2d(in_channels=8,
...                            out_channels=8,
...                            kernel_size=5)
>>> x = convt(y)
>>> x.shape
```

should get the same shape back!

Inverse Convolution + Padding

```
>>> x = torch.randn(2, 8, 64, 64)
>>> conv = nn.Conv2d(in_channels=8,
...                  out_channels=8,
...                  kernel_size=5,
...                  padding=2)
>>> y = conv(x)
>>> y.shape
```

Inverse Convolution + Padding

```
>>> x = torch.randn(2, 8, 64, 64)
>>> conv = nn.Conv2d(in_channels=8,
...                  out_channels=8,
...                  kernel_size=5,
...                  padding=2)
>>> y = conv(x)
>>> y.shape

>>> convt = nn.ConvTranspose2d(in_channels=8,
...                            out_channels=8,
...                            kernel_size=5,
...                            padding=2)
>>> x = convt(y)
>>> x.shape
```

Inverse Convolution + Padding

```
>>> x = torch.randn(2, 8, 64, 64)
>>> conv = nn.Conv2d(in_channels=8,
...                  out_channels=8,
...                  kernel_size=5,
...                  padding=2)
>>> y = conv(x)
>>> y.shape

>>> convt = nn.ConvTranspose2d(in_channels=8,
...                            out_channels=8,
...                            kernel_size=5,
...                            padding=2)
>>> x = convt(y)
>>> x.shape
```

should get the same shape back!

Inverse Convolution + Stride

```
>>> x = torch.randn(2, 8, 64, 64)
>>> conv = nn.Conv2d(in_channels=8,
...                  out_channels=8,
...                  kernel_size=5,
...                  stride=2)
>>> y = conv(x)
>>> y.shape
```

Inverse Convolution + Stride

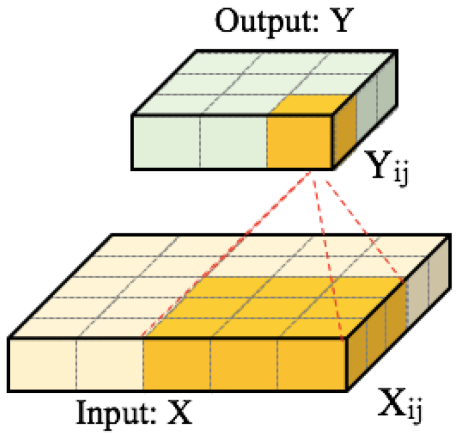
```
>>> x = torch.randn(2, 8, 64, 64)
>>> conv = nn.Conv2d(in_channels=8,
...                  out_channels=8,
...                  kernel_size=5,
...                  stride=2)
>>> y = conv(x)
>>> y.shape

>>> convt = nn.ConvTranspose2d(in_channels=8,
...                            out_channels=8,
...                            kernel_size=5,
...                            stride=2)
>>> x = convt(y)
>>> x.shape
```

Inverse Convolution + Stride

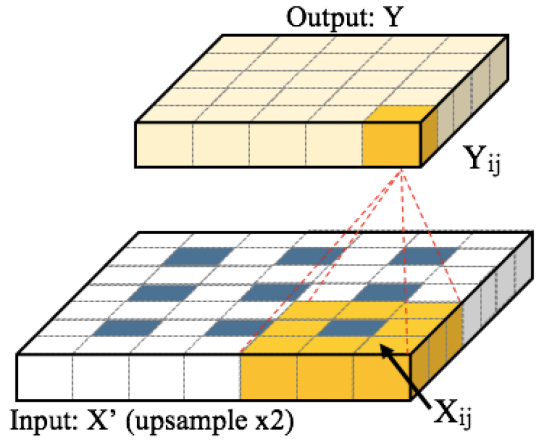
```
>>> x = torch.randn(2, 8, 64, 64)
>>> conv = nn.Conv2d(in_channels=8,
...                  out_channels=8,
...                  kernel_size=5,
...                  stride=2)
>>> y = conv(x)
>>> y.shape

>>> convt = nn.ConvTranspose2d(in_channels=8,
...                            out_channels=8,
...                            kernel_size=5,
...                            stride=2)
>>> x = convt(y)
>>> x.shape
```

. . . almost the same shape . . .

Transpose Convolution Layer



(**a**) Convolutional layer: the input size is $W_1 = H_1 = 5$; the receptive field $F = 3$; the convolution is performed with stride $S = 1$ and no padding ($P = 0$). The output Yis of size $W_2 = H_2 = 3$.



(**b**) Transposed convolutional layer: input size $W_1 = H_1 = 3$; transposed convolution with stride $S = 2$; padding with $P = 1$; and a receptive field of $F = 3$. The output Yis of size $W_2 = H_2 = 5$.

Figure 2: https://www.mdpi.com/2072-4292/9/6/522/htm

More at `https://github.com/vdumoulin/conv_arithmetic`

Output Padding

```
nn.ConvTranspose2d(in_channels=8,
                   out_channels=8,
                   kernel_size=5,
                   stride=2,
                   output_padding=1) # +1 to output
                                     # width/height
```

Autoencoder

To demonstrate ConvTranspose2d, we will build a network that:

- Finds a lower dimensional representation of the image
- Then reconstructs the image from the low-dimensional representation

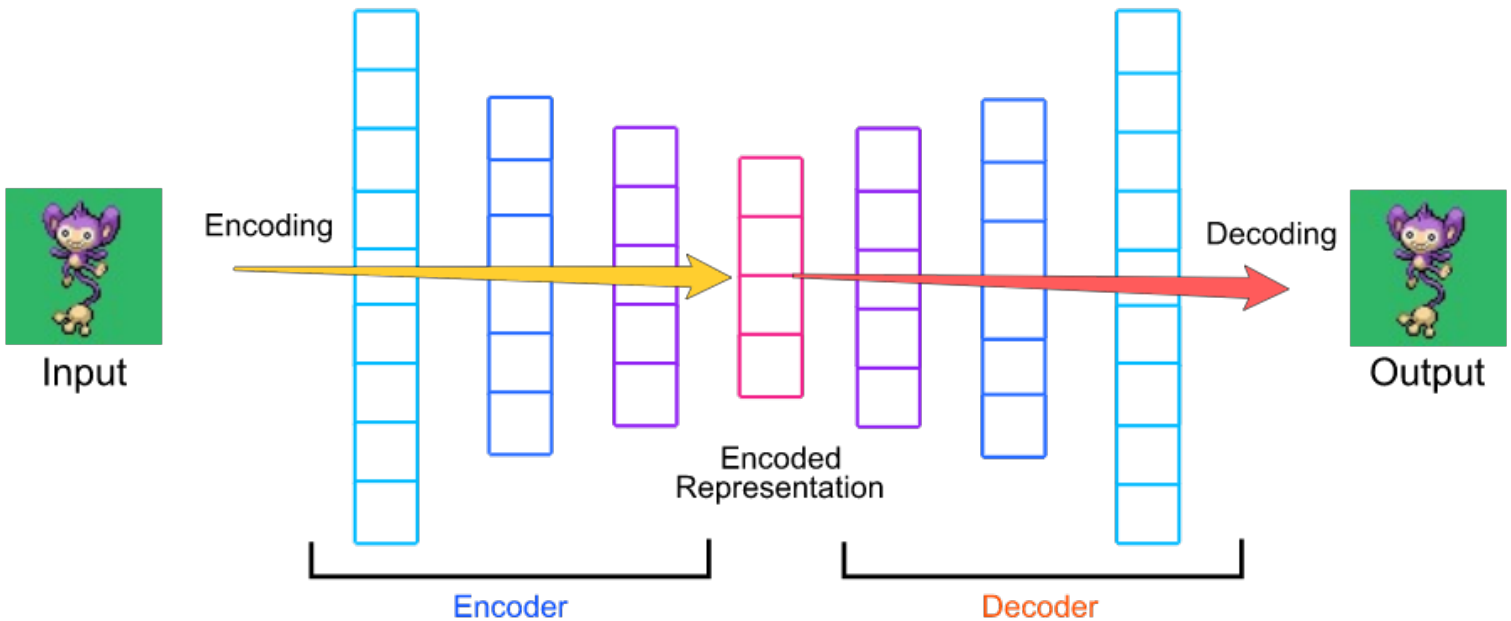
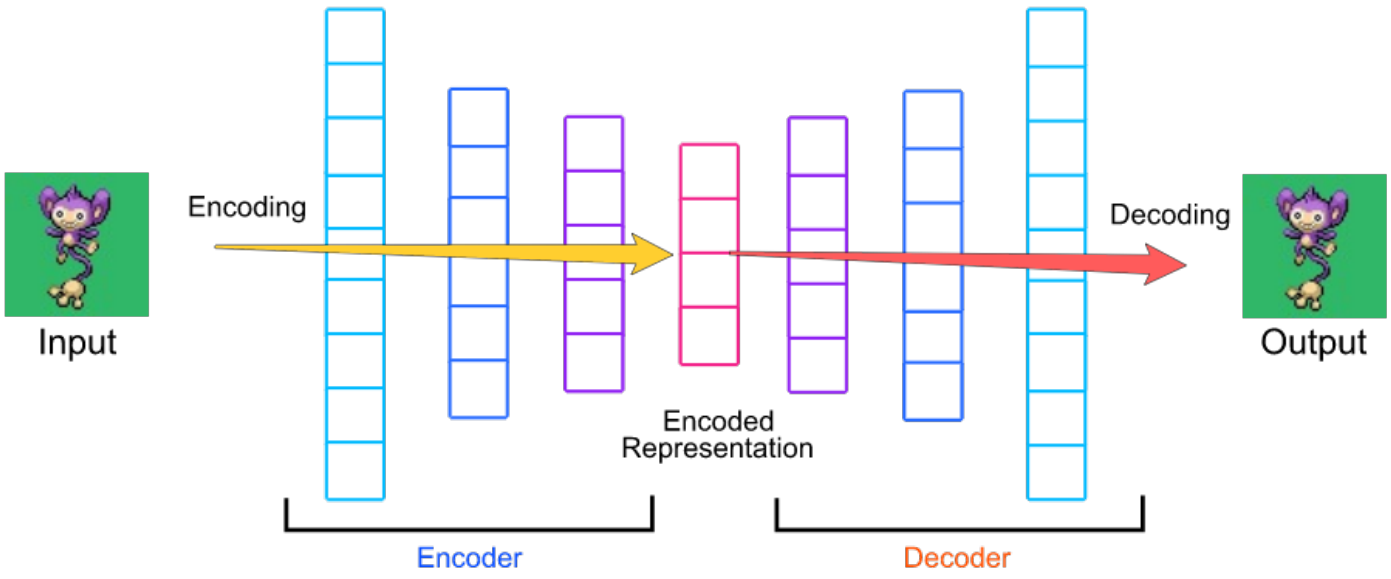


Figure 3: https://hackernoon.com/how-to-autoencode-your-pok%C3%A9mon-6b0f5c7b7d97

The components of an autoencoder



Encoder:

- Input = image
- Output = low-dimensional embedding

Decoder:

- Input = low-dimensional embedding
- Output = image

Why autoencoders?

- Dimension reduction:
  - find a low dimensional representation of the image
- Image Generation:
  - generate new images not in the training set

Autoencoders are not used for **supervised learning**. The task is *not* to predict something about the image!

Autoencoders are considered a **generative model**.

How to train autoencoders?

- Loss function:
  - How close were the reconstructed image from the original?
  - **Mean Sqaure Error Loss**: look at the mean square error across all the pixels.
- Optimizer:
  - Just like before!
  - Introduce a new optimizer: `Adam`
  - Commonly used for other network architectures too
- Training loop:
  - Just like before!

Structure in the Embedding Space

The dimensionality reduction means that there will be structure in the embedding space.

If the dimensionality of the embedding space is not too large, similar images should map to similar locations.

Interpolating in the Embedding Space