Deep Learning

Hassan Khurram

June 2021

Contents

Chapter 1

Neural Networks and Deep Learning

1.1 Week 1

1.1.1 Neural Networks Review

Neural networks for predicting housing prices:

We can use activation functions on neurons such as, $ReLU(z)$, where:

$$ReLU(z) = max(0, z) \tag{1.1}$$

This is used since housing prices can't be -ve.

Peceptron:

size, x ⟶ O ⟶ price, y

Stacking in hidden layer neurons basically represents lower-level features determined by input features. Those hidden features are learned by themselves. But in neural network implementations, the features are "black boxes".

1.1.2 Learning with Neural Networks

- Supervised Learning: Where the inputs x maps to known outputs y.
- Standard NN for real estate and ads
- CNNs for image applications
- RNNs: Audio is represented as a one-dimensional time series (sequence data) and Machine translation / NLP (also sequence data)
- Custom/Hybrid for autonomous driving (maps image, radar info to position of other cars)

Structured data: databases of data, where we know what the feature names are and they have a well defined meaning

Unstructured Data: Refers to raw audio, images, or text. The features may be words or pixels. This is harder to make sense of by computers. But deep learning has advanced its ability.

1.1.3 Why is Deep Learning Taking Off?

Traditional learning algs such as SVMs, logistic regression only converges in performance past a certain amount of labelled data.

Increasing size of the NN means that performance continues to increase with scaled data.

Sigmoid function is slow for learning due to vanishing gradients for high z values (or large -ve values). Moving to the ReLU function makes gradient descent much faster since the gradient of J wrt to parameters is constant (does not vanish).

1.2 Week 2

1.2.1 Binary Classification

We want to use vectored methods over the entire training set rather than using for loops.

Logistic regression is used for binary classification:

$y \in \{1(cat), 0(noncat)\}$

Input is an image represented as a tensor of pixel intensity values taht is 64x64x3 , where the 3 comes from RGB (3 base colors)

Define x as an unrolled column vector with all the elements from the tensor with dimension $\mathbb{R}^{n_x = 64x64x3}$. n_x is the number of features per example.

We're now defining the input feature matrix $X \in \mathbb{R}^{n_x \times m}$ differently, where each example (of an image with pixels unrolled into a column) is a column of X st:

$$X = \begin{bmatrix} | & | & & | \\ X^{(1)} & X^{(2)} & ... & X^{(m)} \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{n_x \times m} \tag{1.2}$$

Stack Y's training examples as columns so that $Y \in \mathbb{R}^{1 \times m}$ is thus a row vector:

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & ... & y^{(m)} \end{bmatrix} \tag{1.3}$$

1.2.2 Logistic Regression

Given x want $\hat{y} = P(y = 1|x)$

Given parameters: $w \in \mathbb{R}^{n_x}$ and $b \in \mathbb{R}$

Output: $\hat{y} = \sigma(w^T x + b\ )$

A linear hypothesis is a bad algorithm because we only want $\curvearrowright$ between 0 and 1 since its a probability. That's why we apply sigmoid.

Unlike in the ML class, we're not using θ_0 for the bias and an additional 'row' of $x_0 = 1$'s for all training examples, since here we separate the bias and weight vectors.

1.2.3 Logistic Regression Cost Function

Just look at the other notes.

$$L(\hat{y}, y) = -\left(ylog\hat{y} + (1-y)log(1-\hat{y})\right) \tag{1.4}$$

Cost function is:

$$J(w, b) = -\frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) \tag{1.5}$$

1.2.4 Gradient Descent

Want to find w, b that minimize $J(w, b)$, a convex function from equation 5.

Initialize w, b to 0 in logistic regression.

Update:

$$w := w - \alpha\frac{dJ}{dw} \tag{1.6}$$

in code the derivative term is "dw"

1.2.5 Computation Graph

Organizes forward pass from left to right to compute J. Its like the partial differentiation tree of all variables and what there dependent on from the left.

1.2.6 Chain Rule visualized with a Computation Graph

dvar is the derivative in code of the final output variable (like J always) wrt that var.

1.2.7 Logistic Regression Gradient Descent

$$\text{"}da\text{"} = \frac{dL(a,y)}{da} = -\frac{y}{a} + \frac{1-y}{1-a} \tag{1.7}$$

$$\text{"}dz\text{"} = \frac{dL(a,y)}{dz} = \frac{dL}{da} \cdot \frac{da}{dz} = a - y \tag{1.8}$$

$$\text{"}dw_1\text{"} = x_1 \cdot \text{"}dz\text{"} = x_1(a-y) \tag{1.9}$$

1.2.8 Gradient Descent on m examples

Implementing gradient descent on m training examples.

$$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial w_i} L\left(a^{(i)}, y^{(i)}\right) \tag{1.10}$$

```
J = 0; dw_1 = 0; dw_2 = 0; db = 0

for i in range(1,m):
    # compute derivatives wrt to each training examples and add them up

    z_i = w' x_i + b
    a_i = sigmoid(z_i)

    J += - [ y_i (log a_i + (1 - y_i) log(1 - a_i) ]

    dz_i = a_i - y_i

    d_w1 += x1_i dz_i
    d_w2 += x2_i dz_i
    db += dz_i


J /= m;
dw_1 /= m ...
```

The derivatives are accumulators over the entire trianing set, or the sum of the individual derivatives wrt to all the params.
Update rules:

```
w_1 = w_1 - alpha * dw_1
...
b = b - alpha * db
```

1.2.9 Vectorization

Vectorization is better than using for loops.

1.3 Week 3

1.3.1 Neural Networks Forward Prop

$z^{[1]} = W^{[1]}x + b^{[1]}$

$a^{[1]} = \sigma(z^{[1]})$

$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$

$a^{[2]} = \sigma(z^{[2]})$

1.3.2 Vectorizing

$a^{[2](i)}$ = output neuron activation for layer 2 example i

Instead of using a for loop over all m training examples, we can take advantage of X with each example stacked as a column.

$Z^{[1]} = W^{[1]}X + b^{[1]}$

$A^{[1]} = \sigma(z^{[1]})$

$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$

$A^{[2]} = \sigma(z^{[2]})$

1.3.3 Activation functions

Sigmoid function. Can be used for binary classification in the output layer.

tanh function. Centers the data at zero and so its much better than the sigmoid function for hidden layers.

$$g(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-1}} \tag{1.11}$$

But both of the activation functions above face vanishing gradients for large (or large negative) z values.

ReLU function (rectified linear unit):

$$a = max(0, z) \tag{1.12}$$

NN learns faster since the slope of the function doesn't go to zero on both sides of the domain and since most hidden layers will have $z > 0$.

Leaky ReLU:

$$a = max(0.01z, z) \tag{1.13}$$

Why do we need non-linear activation functions?

If we didn't have non-linear activation functions, then $\hat{y}$ would only be a a linear function of x.

$$a^{[2]} = (W^{[2]}W^{[1]})x + (W^{[2]}b^{[1]} + b^{[2]}) \tag{1.14}$$

It's just a linear activation function. We might as well have no hidden layers. Linear hidden layers are useless. But linear activation functions may be used in the output for say, housing prices, a real number without bound (even then use ReLu). But still use a non-linear function in the hidden layers.

1.3.4 Gradient Descent

Refer to ML course notes.

1.4 Week 4

- L = number of layers
- $n^{[l]}$ = number of units in layer l
- $a^{[l]}$ = vector of activations in layer l
- $a^{[l]} = g^{[l]}(z^{[l]})$
- $W^{[l]}$ = weights for $z^{[l]}$

1.4.1 Forward Propagation in Deep NNs

We can vectorize the implementation over all training examples stacked horizontally. But we cannot vectorize (remove for loop) from looping over all the layers.

$$Z^{[1]} = W^{[1]}X + b^{[1]} \tag{1.15}$$

$$(n^{[1]}, m) = (n^{[1]}, n^{[0]}) \cdot (n^{[0]}, m) + (n^{[1]}, 1) \tag{1.16}$$

So the dimensions work out in batch gradient descent vectorized. The b vector is expanded over all m examples by Python *NumPy broadcasting.*

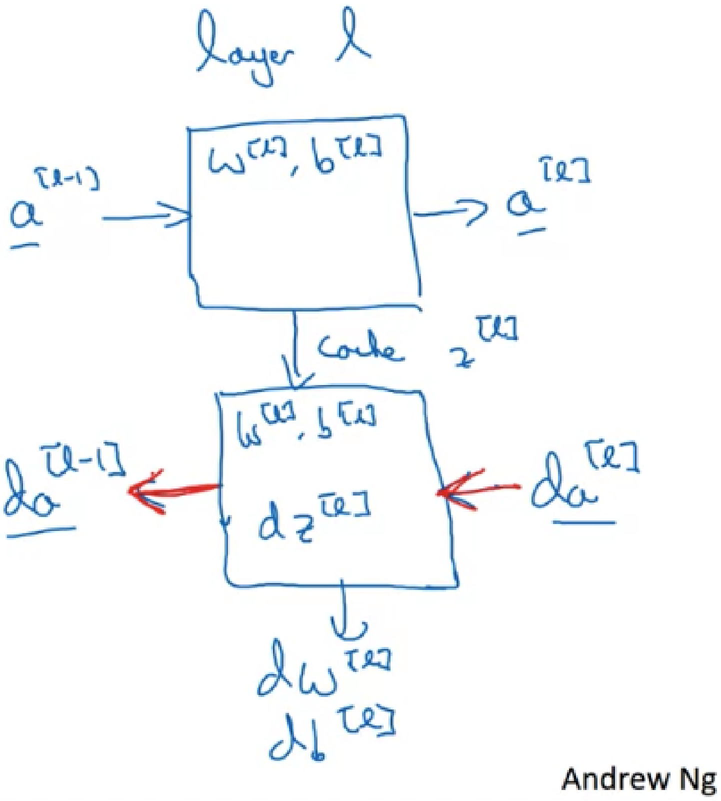

Figure 1.1: Forward and Back-prop

1.4.2 Why Deep Networks?

With more deeper layers, pixels can group together to allow for edge detection, eventually to face detection represented by more complex functions (e.g.). Low level to higher level features as deeper layers progress.

Circuit theory and deep learning

There are functions you can compute with a small L-layer deep nueral network that shallower networks require exponentially more hidden units to compute. Number of hidden neurons in shallow NNs rises as $\mathcal{O}(2^n)$.

1.4.3 Implementation

Forward prop for layer l:

Input $a^{[l-1]}$

Output $a^{[l]}$, cache $z^{[l]}, W^{[l]}, b^{[l]}$

Backward prop for layer l:

Input $da^{[l]}$

Output $da^{[l-1]}, dW^{[l]}, db^{[l]}$

1.4.4 Parameters vs Hyperparameters

- Parameters are the weights and basis matrices.
- Hyperparameters include learning rate, number of iterations, number of hidden layers, number of hidden units, choice of activation functions. Later: momentum, mini-batch size, regularization term.

Chapter 2

Improving Deep Neural Networks

2.1 Week 1

2.1.1 Train / Development / Test sets

Applied deep learning is an iterative process.

- *Structured Data:* Ads, search, security, logistics

- Intuitions from one area might not translate well to other areas. Considerations also include if we're using GPUs or CPUs or combinations of them.

- Train algorithms on the training set. Then use the hold-out / cross validation / development set to determine which algorithm performs best, then test the final algorithm on the test set (for an un-biased estimate of performance).

- In the modern big data era: If we have 1 M total examples in the dataset, 10,000 examples is sufficient for the dev and test sets.

There may be a distribution mismatch between the training set and the dev/test sets (eg. training set consists of cat pictures from web-pages, while the dev/test set comes from user taken cat photos). We only need to make sure that the **dev** and **test** sets come from the same distribution.

2.1.2 Bias and Variance

There is less of a bias-variance tradeoff in the deep learning era.

- High bias: underfitting

- "Just right": In-between

- High variance: overfitting (to the training set)

The metrics we use are **training set error,** and the **dev set error**. The Bayes error is the optimal error rate (approximated as human error). Another assumption is that the distribution of the training set and the dev set is similar.

2.1.3 Recipe for ML

- High bias (underfitting)? (training data performance): ⟶ Bigger network (more layers), train longer, optimization algorithms (Adam), NN architectures.
- High variance? (dev set performance): ⟶ More data, regularization, NN architectures.

Note: So long as we can maintain a bigger network and more data, neither bias or variance worsens. This is why there's less of a bias/variance trade-off. Regularization to address high variance does involve a little trade-off in that the bias could increase (when having a smoother decision boundary etc.).

2.1.4 Regularization

To address the high variance problem.

2.1.5 For Logistic Regression

L2 Regularization:

$$J(\mathbf{w}, b) = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}||\mathbf{w}||_2^2 \tag{2.1}$$

$$||\mathbf{w}||_2^2 = \mathbf{w}^T\mathbf{w} \tag{2.2}$$

L1 Regularization

We instead add $\frac{\lambda}{2m}\sum_{i=1}^{n_x}|\mathbf{w}|$. $\mathbf{w}$ will be sparse.

λ = regularization parameter (a hyper-parameter tuned from testing on the dev set)

2.1.6 Neural Network

We add this to the cost function:

$$\frac{\lambda}{2m}\sum_{l=1}^{L}||W^{[l]}||_F^2 \tag{2.3}$$

Where the **Frobenius norm** of a matrix is defined as:

$$||W^{[l]}||_F^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} \left(W_{ij}^{[l]}\right)^2 \tag{2.4}$$

L2 regularization is also called **weight decay**.

```
W[l] = W[l] - alpha ( [from backprop] + (lambda / m) W[l] )
```

The weight before being iterated is multiplied by $1 - \alpha\frac{\lambda}{m}$, which is between 0 and 1.

- The intuition behind regularization is that if λ is set to a very high number, the weights will be much closer to zero, so that much of the impact of the hidden layers are zeroed out. We would get the high bias case.
- This is because gradient descent tries to minimize the cost. If λ inflates this cost, then to compensate for this in training, the magnitude of weights would be smaller so the impact of hidden layers is lessened. This is almost like reducing the size of the effective network.
- Alternative view: In the tanh() activation function, if the z values are smaller since the weights are penalized to be small, then $g(z)$ will be roughly linear (close to linear regression for even a deep network), or less complex.

2.1.7 Dropout Regularization***

We set some probability of eliminating (zeroing out) certain nodes in every layer. Typically $P = 0.5$, which is the chance to keep nodes, and remove nodes. All links to removed nodes are removed, and we get left with a diminished network. We repeat and get a slightly different randomly chosen reduced network.

Inverted Dropout

```
keep_prob = 0.7
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
a3 = np.multiply(a3, d3)
a3 /= keep_prob
```

d3 is a vector, boolean array generated from drawing numbers randomly, of the same shape as $\mathbf{a}^{[3]}$ and is either 0 (eliminating node) or 1 (keeping). The last line is carried out so that the expected value of $\mathbf{a}^{[3]}$ is corrected back to its original value after some nodes are zeroed out. DO NOT use regularization when testing.

Second intuition: Can't rely on any one feature, so have to spread out weights. A single unit can't rely on any one input, since that input could be eliminated. This has the effect of shrinking weights. Keep-prob is lower for some layers that are larger (larger overfitting risk).

Dropout is very useful in computer vision.

2.1.8 Other ways of regularization

- Data augmentation (horizontal flipping, color shift, random crops of image)
- Early stopping: dev set decreases before increasing again w/ number of iterations. Stop before dev set error increases. (midsize norm of w, less overfitting)

2.2 Normalizing Inputs

Subtract mean (zero out the mean):
$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$
$x := x - \mu$
Normalize variance:
$\sigma^2 = \frac{1}{m} = \sum_{i=1}^{m}(x^{(i)} - 0)^2$
$x := x/\sigma$

This normalization is done feature by feature: we center the mean of each feature in each example to zero, and divide by that feature's standard deviation (Z-score normalization)

We use the same μ and σ determined in the training set for normalizing test set examples. Spherical contours over normalized features makes **gradient descent steps faster**.

2.3 Vanishing / Exploding Gradients

If $W^{[l]} > I$ then the activations increase exponentially.
If $W^{[l]} < I$ then the activations decrease exponentially.
This is a problem when using tanh() or sigmoid()

2.3.1 Weight Initialization - He Initialization

Set the $Var(W_i) = \frac{1}{n}$

```
W[l] = np.random.randn(shape(W[l])) * np.sqrt(2 / n[l-1])
```

If we have a lot of neurons in the previous layer, set randomly initialized weights individually to smaller values.

2.3.2 Gradient Checking

- Don't use in training, only to debug
- Does not work with dropout since J changes each time

2.4 Week 2: Optimization algorithms

2.4.1 Mini-batch gradient descent

- Refer to ML notes.

- Even vectorization can be slow if we have millions of example before taking a single step of gradient descent.

- Split the data into mini-batches of size, say 1000.

- Batches: $X^{\{1\}}, X^{\{2\}}...X^{\{t\}}$

- epoch: single pass through the training set. In mini-batch, one epoch involves multiple updates.

- Stochastic Gradient Descent: If the mini-batch size is 1 (each example is its own mini-batch) since we look at one example only before updating parameters.
- Batch Gradient Descent: If the mini-batch size is m (entire training set before updating).

How do we choose a mini-batch size? It depends on the size of the training set. If we have a small training set, use batch gradient descent. Typical mini-batch sizes are 64 ... 512. Make sure mini-batch fits in CPU/GPU memory.

2.4.2 Exponentially Weighted Moving Averages

$$v_0 = 0$$
$$v_1 = \beta v_0 + (1 - \beta)\theta_1$$
$$\vdots$$
$$v_t = \beta v_{t-1} + (1 - \beta)\theta_t$$

v_t is approximating over $\approx \frac{1}{1-\beta}$ days' temperature.

For computing the value at the current t, we take the weighted sum over the previous v_t values with coefficients multiplied with the 'v' values (with decreasing t) decaying to zero as we go, say back in time.

```
v_theta = 0

repeat {
    Get next theta_t
    v_theta := beta * v_theta + (1 - beta) theta_t
}
```

This is a very efficient method.

Bias Correction in Exponentially Weighted Averages

Instead of initializing 'v' at around 0 (with the first few terms being close to zero too), we use a bias correction term. Take:

$\frac{v_t}{1-\beta^t}$

2.4.3 Better than gradient descent? Gradient Descent with Momentum

- Compute an exponentially weighted average of gradients over the previous iterations.

Momentum:

```
On iteration t:
    Compute dW, db on current mini-batch
    v_dW = beta * v_dW + (1-beta) * dW
    v_db = beta * v_db + (1-beta) * db
    W = W - alpha * v_dW
    b = b - alpha * v_db
```

This smoothens / averages out the previous gradients computed when iterating on the current iteration t. Hyperperameters are α, β.

2.4.4 RMSprop - Root Mean Square prop

Exponentially weighted averages of the squares of the gradients.

```
On iteration t:
    Compute dW, db on current mini-batch
    S_dW = beta * S_dW + (1-beta) * dW**2
    S_db = beta * S_db + (1-beta) * db**2
    W = W - alpha * dW / sqrt(S_dw)
    b = b - alpha * db / sqrt(S_db)
```

Say we want learning in the vertical direction to be slow and that for the horizontal direction to be large to converge to the minimum. This dampens out the oscillations when J is converging to a minimum.

2.4.5 Adam (adaptive moment estimation) Optimization Algorithm - Combination of Momentum and RMSprop

```
set all to zero


On iteration t:
```

```
Compute dW, db on current mini-batch

# momentum terms being averaged over - first moment
v_dW = beta1 * v_dW + (1-beta) * dW
v_db = beta1 * v_db + (1-beta) * db

# RMSprop terms being averaged over - second moment
S_dW = beta2 * S_dW + (1-beta) * dW**2
S_db = beta2 * S_db + (1-beta) * db**2


# bias correction for momentum terms
V_dW_corrected = v_dW / (1 - beta1^t)
V_db_corrected = v_db / (1 - beta1^t)

# bias correction for RMSprop terms
S_dW_corrected = s_dW / (1 - beta2^t)
S_db_corrected = s_db / (1 - beta2^t)


# update rule:

W = W - alpha * V_dW_corrected / (sqrt(S_dW_corrected) + epsilon)
b = b - alpha * V_db_corrected / (sqrt(S_db_corrected) + epsilon)
```

2.4.6 Learning Rate Decay

$$\alpha = \frac{1}{1 + decay\,rate * epoch\,number}\alpha_0 \tag{2.5}$$

2.4.7 Summary

Regularization is a modification to the cost function.

Adam, momentum and RMSprop are alternatives to GD in that they are modifications to the GD update rule.

2.5 Week 3

Hyperperameter: $\alpha, \beta, \beta_1, \beta_2, \epsilon$, number of layers, hidden units, learning rate decay, mini-batch size.

- Try randomly set of points. We could also use a 'course to fine' scheme: randomly sample points, then search and sample more points over a smaller square (more dense search).
- Use an appropriate scale to pick hyperparameters: A uniformly random sampling can work on number of layers, number of hidden units.
- To search for an optimal α, we use a log scale to sample instead.

```
r = -4 * np.random.rand() # r is between -4 and 0
alpha = 10**r
```

- For β when using only momentum: we also use a log scale.

Why dont we use a linear scale for α, β? Since as we approach 1, the hyperparameters become more sensitive. Look at the learning curves to see how effective hyperparameters are.

2.5.1 Batch Normalization

This is where we normalize activations in a hidden layer, so as to train the weights and biases to the enxt layer faster.

Practice: normalize $z^{[l]}$

Implementing Batch Norm:

Given some intermediate values in NN: $z^{(1)} ... z^{(m)}$

$\mu = \frac{1}{m} \sum_i z^{(i)}$

$\sigma^2 = \frac{1}{m} \sum_i (z_i - \mu)^2$

$z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$

$\tilde{z}_{norm}^{(i)} = \gamma z_{norm}^{(i)} + \beta$

Now, the vector of weighted sums in one layer will be normalized: with a mean of zero and variance of one. We can modify this normalized value to have a non-zero mean (useful for ReLU or other activation functions).

Because batch-norm zeros out the mean, we can remove the biases. We replace $b^{[l]}$ with $\beta^{[l]}$.

Why does batch norm work?

- We are now normalizing the hidden units instead of just the input features. It makes changes to weights in deeper layers more robust.

Covariate shift: If we learn some $X \longrightarrow Y$ mapping on one distribution of data, but the input distribution is shifted on the test set.

- Batch norm reduces how much the previous layer's activations shift around after the weights and biases are updated in those previous layers. (mean and variance stay constant) This makes learning easier.

Batch norm as regularization:

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch, the mean/variance may be noisy.
- This also adds some noise to $z^{[l]}$ values in that mini-batch. Hence adds some noise to activations in the hidden layers.
- Slight regularization effect (can't rely on any one neuron in the previous layer as it may change due to noise)

At test time: Obtain μ, σ^2 as estimates using exponentially weighted averages (across mini-batches).

2.5.2 Softmax Regression

Generalization of logistic regression for multiclass classification, with C classes in total (equal to the total number of units in the output layer).

To get an output vector of probabilities per class (so total sum of outputs =1), use the softmax activation function:

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{j=1}^{C} e_i^{z^{[L]}}} \tag{2.6}$$

Loss function:

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{4} y_j log(\hat{y}_j) \tag{2.7}$$

(maximum likelihood estimation from statistics)

Chapter 3

Convolutions Neural Networks

3.1 Week 1

Say we have a $1000 \times 1000 \times 3$ dimensional image. If we use a fully connected network, the dimension of $W^{[1]}$ would be $(1000, 3\,million)$, which totals up to a BILLION parameters! This is infeasible because we couldn't possibly have enough data to prevent over fitting due to the exorbitant number of parameters we have. To solve this problem, we instead make use of convolution operations.

3.1.1 Edge detection and convolutions

Early layers detect edges, before objects and complete objects (faces), or more higher level features in deeper layers.

How do we detect vertical edges?

We construct a 3×3 filter / kernel that convolves with the input image matrix.

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \tag{3.1}$$

Difference between light to dark vs dark to light edges. How do we not hard code this?

Sobel Filter:

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \tag{3.2}$$

Schorr Filter:

$$\begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix} \tag{3.3}$$

Or we can learn the 9 parameters as the elements of the 3 by 3 filter as part of a good edge detecting algorithm. That way, any orientation of images can be learned for the filters involved.

3.1.2 Padding

- One downside is that the matrices shrink every layer a convolution is applied. . .
- Throwing away information from the edges. This is since we don't use the edge pixels all too often compared to the pixels in the center or away from edges. . .

We can pad the image with a border of 0's. This way, we preserve the original dimensions of the image (the dimensions before padding) after convolving once.
$p =$ padding amount.

$$n + 2p - f + 1 \times n + 2p - f + 1 \tag{3.4}$$

Now we no longer have less information from the corner pixels in the output matrix.

Valid and Same convolutions:

- Valid: No padding
- Same: Pad so that the output size is the same as the input size. $p = \frac{f-1}{2}$ ensures that this condition holds.

  btw: f is usually odd. We want to have a central pixel.

3.1.3 Strided Convolutions

Stride is how big a step the convolution box takes on the input image. How much it ”hops” is defined as the stride.

Output dimensions by taking into account stride are:

$$\left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor \tag{3.5}$$

But the convention is that the convolution box lies entirely in the image.

Cross correlation vs convolution: in ML, we don't bother with flipping and reflecting both horizontally and vertically the filter.

3.1.4 Convolutions Over Volume

Convolutions on RGB images

If the image is a 3D tensor, so will the kernel. The filter has the same number of channels as the image (input).

The output is a 2d matrix.

How does this work?

WE still take the hadamard product of the 3d kernel tensor and the input image, and then take the sum to get the output. This is the reason why we get a 2D output if we use only one filter since all the entries in the output are $\in \mathbb{R}$.

Multiple Filters

If we want to detect multiple forms of edges, we're gonna have to use multiple filters.

We take the two filter outputs and then stack it so that the output volume is say, 44×2. The 2 cmoes from the fact that we used 2 different filters.

Summary:

$n \times n \times n_c * f \times f \times n_c \longrightarrow n - f + 1 \times n - f + 1 \times n_c'$

Where n_c' is the number of filters.

3.1.5 One layer of a convolutional network

We add a bias $b_i \in \mathbb{R}$ and apply the non-linearity like $ReLU(X * F)$, to the output matrix. Then we stack these matrices into a tensor.

Here, the input image can be anything. The number of parameters will remain fixed. For the sake of lower computational complexity on orders of magnitude, this is well preferred over having a billion features that would be fully, or densely connected to the input image if it was just flattened to a column vector, which would be the case if we just used traditional neural networks. By making use of **sparse connections**, we effectively prevent overfitting by allowing a NN that maintains the same number of parameters.

Notation:

- If layer l is a convolutional layer.
- $f^{[l]}$ = filter size
- $p^{[l]}$ = padding
- $s^{[l]}$ = stride

- $n_c^{[l]}$ = number of filters = the depth of the output tensor

Dimensions of tensors involved:

- Each filter is: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$
- Activations: $a^{[l]} \longrightarrow n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$
- Vectorized over all m examples: $A^{[l]} \longrightarrow m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$
- All Weights (all the filters put together): $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$
- Bias: $n_c^{[l]}$ But its still $(1, 1, 1, n_c^{[l]})$

Input: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$
Output: $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$
Output volume size:

$$n_H^{[l]} = \left\lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor \tag{3.6}$$

When applying convolutions over multiple layers, at the end, we have to unroll, or flatten it into a column vector and taking the logistic / softmax function over all of the neurons to get $\hat{y}$

We also have *pooling layers* and *fully connected layers.*

3.1.6 Pooling layers

Max pooling: Take the input and break it into a smaller matrix. We split the input matrix into blocks of smaller matrices and we take the max from each of these blocks. Think of the input as a hidden layer of features, where only the max features are preserved in the output of max pooling. Gradient descent has nothing to learn and change in max pooling. Input depth is equal to the output depth.

Note: Andrew Ng recurrently mentions how a conceptual basis about these concepts would make you better prepared to understand the "literature" on AI advances, by which he refers to papers and research. :thonk:

3.1.7 Why Convolutions

The number of parameters and thus the time complexity and memory requirements of the algorithm are vastly reduced. It allows for translation invariance, or **equivariance** to translation.

Parameter Sharing: A feature detector (such as vertical edge detectors) that's useful in one part of the image is probably useful in another part of the image. An element of the kernel is multiplied by almost all pixels in the input image, for example.

Sparsity of Connections: In each layer, each output value/unit depends only on a small number of inputs. So one output entry in the output matrix per filter depends only on a ”number of features per filter” number of inputs.

$$Cost\ J = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) \tag{3.7}$$

Then apply gradient descent, Adam, or momentum.

3.2 Week 2

- Classic networks: LeNet-5, AlexNet, VGG
- ResNet - Residual network (152 layer NN)
- Inception case study

3.2.1 Classic Networks

LeNet-5 - LeCun

`http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf`
The goal was to recognize hand-written digits. $(32 \times 32 \times 1)$ inputs

- **Layer 1**

  First step: Use set of 6 5×5 filters with a stride of 1, without any padding. Size of output reduced to $28 \times 28 \times 6$.

  Second step: Apply average pooling with $f = 2$, $s = 2$. Dimensions reduced to $14 \times 14 \times 6$.

- **Layer 2**

  Third step: Apply 16 filters that are each 5×5.

  Fourth step: Average pool (see 2nd step)

- **FC Layers 3 and 4**

  Fifth step: Flattened, fully connected layer. Then another FC layer before computing $\hat{y}$.

In the modern era, we use way bigger algorithms than this network. We also use a softmax activation on the output neuron, and most commonly use max pooling over average pooling. General form is still CONV POOL CONV POOL FC FC OUTPUT. Another distinction is that sigmoid and tanh activation functions were used on each convolutional layer, and that given lower computational complexity supported in that era (1998), the filter windowed through selected channels for certain pixels rather than going through every pixel in all channels.

AlexNet (UofT)

`https://www.cs.toronto.edu/~kriz/imagenet_classification_with_deep_convolutional.pdf`
- 60M parameters
- Multiple GPUs
- Local Response Normalization: for each position in (width, height), over all channels, we normalize the parameters. However, this was not all that useful.

VGG - 16

CONV = 3×3 filter, $s = 1$, same

MAX-POOL = 2×2, $s = 2$

General pattern is common to all: that n_h, n_w go down, and n_c go up.

3.2.2 Residual Networks (ResNets)

The point of ResNets is to make the machine learning practitioner be less hesitant to add in more layers. This is because in plain neural nets, performance diminishes due to vanishing gradients and increasing computational complexity past a certain point of the number of layers. Using ResNets (skip connections), one of the blocks can easily learn an identity function *if* the number of layers is too "high" to learn an underlying mapping from $\mathbf{x}$ to $\mathcal{H}(\mathbf{x}) = \hat{\mathbf{y}}$, so performance won't diminish.

We make use of skip connections: taking the activation from one layer, and feeding it to a layer that is much deeper in the network.

Residual Block:

$$a^{[l+2]} = g\left(z^{[l+2]} + a^{[l]}\right) \tag{3.8}$$

The shortcut/skip connection is added before the ReLU activation function. The Keras functional API is used in this context in lieu of the Sequential API, which only takes in one tensor input and outputs one tensor output.

Having a plain network results in an increasing training error eventually, if the network is too deep. This isn't the case in ResNets, where the training error continues to decrease regardless of the number of layers.

$$X \longrightarrow \text{Big NN} \longrightarrow a^{[l]}$$
$$X \longrightarrow \text{Big NN} \longrightarrow a^{[l]} \longrightarrow FC \longrightarrow FC \longrightarrow a^{[l+2]}$$

The identity function is easy for Residual blocks to learn. That is, if we assume L2 regularization. This means that $a^{[l+2]} = a^{[l]}$. Hence adding the residual block with extra layers does not hurt performance. But can we do even better than learning the identity function? Note: *use "same" convolutions to preserve dimensions in this short circuit.*

3.2.3 Inception Networks

We do everything: we use all sizes of convolution kernels and pooling sizes. Stack the outputs of convolving different sized kernels with the input image as the output image. To make all the dimensions match up, we make use of the "same" padding for max pooling as well, with $s = 1$.

- **The problem of computational cost:** We can go from 120 M multiplications to a number that is 10-fold lower using 1×1 filters rather than 55 filters.

- **Bottleneck layer:** Where we make use of 1×1 kernels to shrink the representation (shrink the depth) to lower the computational cost before proceeding to use 5×5 kernels in the subsequent layers.

3.2.4 MobileNet

This is typically used in mobile phone applications.

- Low computational cost at deployment

- Useful for mobile and embedded vision applications

- Normal vs. depth-wise separable convolutions.

1. **Normal Convolution:** Computational cost = filter params × # filter positions × # of filters

2. **Depthwise Seperable Convolution:** Input $\longrightarrow$ Depthwise convolution $\longrightarrow$ Pointwise convolution $\longrightarrow$ output

   (a) **Depthwise:** We treat all of the RGB channels seperately to compute the convolutions.

   (b) **Pointwise:** Then take the (n_{out}, n_{out}, n_C) output tensor and convolve it with a filter that is $(1, 1, n_C)$ to get the final output, convolving using n'_C number of filters.

General improvement in computational complexity ratio:

$$\frac{1}{n'_C} + \frac{1}{f^2} \tag{3.9}$$

MobileNet v2

- Expansion

- Depthwise

- Pointwise (projection)

Also uses a residual connection. The bottleneck block is repeated 17 times, and allows for a richer set of connections / parameters to learn, while keeping the number of activations from layer to layer low (given memory constraints).

3.2.5 Data Augmentation

In almost all cases for machine learning, having more data is beneficial.

- Mirroring
- Random cropping: Taking random samples from a single image (crops).
- Rotation
- Shearing (distort it into a parallelogram)
- Local warping ...
- Color shifting: Change the R, G, B channels with distortions. These shifting values are drawn from a probability distribution (which one?). This makes the learning algorithm more robust against color changes.
- Advanced ways to draw colors. PCA - principle component analysis (AlexNet Paper). Also known as ”PCA color augmentation”. PCA was used in the ML course to reduce the dimensionality of datasets.

Implementations: Images from the harddisk are pulled to a CPU thread where image distortions are performed on a mini-batch, before these images are sent to training (CPU / GPU). The thread and training can run in parallel.

3.2.6 Notes - High-level view of the state of computer vision

Image recognition (binary) vs object detection (multi-class)

Two sources of knowledge:

- Labeled data (x, y)
- Hand engineered features/network architecture/other components

We develop complex hand-engineered networks in the absence of a lot of data. Transfer learning is also useful in this case.

A multi-channel approach is needed to learn invariances other than just translations. Transformations to inputs such as rotations are included in this case. For just translations locally in each image, max-pooling naturally is invariant to them, since in each neighborhood of the input tensor, any shift of say edges or corners detected in a region will still be detected in that general region.

Increasing stride reduces computational and memory complexity.

3.2.7 Quiz / Programming Exercise Errors:

- Skip-connections help the model learn an identity mapping (assuming L2 regularization), NOT a complex non-linear function.
- Be careful with the definition of a "valid pooling"
- **Depthwise Seperable Convolutions:** No, each filter only convolves with the respective color channel in the input image tensor. Each individual 2D filter DOES NOT convolve with all color channels in the input image. The number of filters used in the depthwise step is only equal to the number of color channels in the input image, or $n'_c = n_c$
- Valid padding for convolutions means no padding on the input image at all. In this case, the output of convolution is ALWAYS a tensor with lower values for n_h and n_w compared to that of the input image. The output dimensions are NOT $n_{out} \times n_{out} \times_c$!
- **What does *prefetch()* do and why is it useful?** It "prevents memory bottlenecks when reading from disk".
- **Dropout:**
- **Global average pooling:**

3.2.8 Research Papers' Notes

MobileNetV2: Inverted Residuals and Linear Bottlenecks

- Mobile models to object detection, and mobile **semantic segmentation** models: the process of "linking each pixel in an image to a class label like say all objects as car objects." (Mwiti)
- Algorithm based on an inverted residual structure where shortcut connections are between the thin bottleneck layers and intermediate lightweight **depthwise convolutions**. Much less computationally expensive as it makes use of faster cache memory.

Deep Residual Learning for Image Recognition

- Counterintuitive phenomena: Degradation occurs with an added (excess) number of layers. We would guess that adding excess layers could learn to become identity mappings and so in theory, the deeper model's training error should not exceed that of the shallower model. *But*, solvers have difficulties in approximating identity mappings when using vanilla stacked nonlinear layers.
  - Since the training error also increased with added layers (without skip connections), over-fitting to the training set is not the issue.

  - This optimization difficulty is likely NOT due to vanishing gradients because forward propagated signals were batch normalized (BN).
  - **Possible cause:** ”Deep plain nets may have exponentially low convergence rates.”

- Residual blocks can learn an identity mapping more easily: ”it would be easier to push the residual to zero than to fit an identity mapping by stacking of nonlinear layers.”
- General case where the dimensions of $\mathbf{x}$ and $\mathcal{F}$ don't have to match up: $\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + W_s\mathbf{x}$. W_s is a linear projection of the shortcut (basically a convolution to match dimensions without relu applied).

ImageNet Classification with Deep Convolutional Neural Networks

3.3 Week 3

3.3.1 Object Localization

- The algorithm is also responsible for locating a box around the image it detects and classifies.
- **Detection:** Detect multiple objects in an image.
- **Classification with localization:** Make the ConvNet output both a softmax output for classification, and a bounding box output, (b_x, b_y, b_h, b_w). The midpoint of the object is (b_x, b_y). *We assume that there's only one object in the image.*

Define the class / bounding box label:

p_c = probability that there is an object in the image.

$$\mathbf{y} = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} \tag{3.10}$$

$$L(\hat{\mathbf{y}},\ \mathbf{y}) = (\hat{y_1} - y_1)^2 + ... + (\hat{y_8} - y_8)^2 \quad \text{if } y_1 = 1 \tag{3.11}$$

In practice, we can use a log likelihood error (logistic regression loss) for the c_i terms.

3.3.2 Landmark Detection

Output layer can be (if we assume 64 landmarks on say someone's face): (Instagram filters use this)

$$\mathcal{H}(\mathbf{x}) = \begin{bmatrix} \text{face ?} \\ l_{1,x} \\ l_{1,y} \\ \vdots \\ l_{64,x} \\ l_{64,y} \end{bmatrix} \tag{3.12}$$

These key landmarks are used to localize not just a single object, but features such as the "corner of the eyes", "jawline" and so on.

3.3.3 Object Detection

Sliding windows detection algorithm: The window at each position returns a cropped version of the image that is fed into the ConvNet before the output is 'whether there is a car in the cropped window or not'. The size of the window is increased after sliding through the entire image. The obvious issue with this method is the high computational cost. Also, if we increase the stride, the increased granularity would hurt performance. Lower stride would increase computational cost.

Convolutional implementation of sliding windows

- *Turning FC layer into convolutional layers:* Use a convolution the same size as the input tensor in that layer (with the same number of channels by definition). The output of the 'FC convolution' is $1 \times 1 \times n'_c$, instead of a FC vector $\in \mathbb{R}^{n'_c \times 1}$.

- Allow the different forward passes to share a lot of computation. Instead of running convolution on different subsets of the image, we instead input the entire image. All the predictions are made at the same time from just one pass.

But the position of the bounding boxes are not very accurate.

Bounding box predictions - YOLO

- We carry out image classification and localization for each grid cell in the original image.

- YOLO takes the midpoint of each of the objects, and assigns the object the grid cell that contains that midpoint.

- Because we have 3×3 number of grid cells, the target output is going to be $3 \times 3 \times 8$.

- We get a more precise bounding box for each object. No longer are the sizes of the bounding boxes limited to the uniform sizes of sliding windows.
- A convolutional implementation.
- But what if we have multiple objects per grid? How do we choose the size of the grids?

Specifying the bounding boxes: The convention is that the upper left hand corner of the *single grid cell* is $(0, 0)$. We only find the bounding box dimensions and position relative to the grid cell only.

Intersection over union

It computes the intersection over union of the two bounding boxes. Computes the size of the intersection / size of the union. Correct if $IOU \geq 0.5$. IOU is a measure of the overlap between two bounding boxes.

$$IoU = \frac{B_1 \cap B_2}{B_1 \cup B_2} \tag{3.13}$$

Non-max suppression

Algorithm might find multiple detections of the same object. Multiple grid cells might detect the same object. We want to make sure that we only determine one image once.

It looks at the probabilities p_c of its detection for each object. We only highlight the bounding box with the highest probability. All the remaining rectangles that overlap alot (have a high IOU) with the highlighted rectangle are then suppressed (why? since having a high IOU means that the other rectangles are close to, and thus associated with the object too, but have lower p_c values).

1. Discard all boxes with $p_c \leq 0.6$, or that were assigned a low probablity
2. While there are any remaining boxes:
   (a) Pick the box with the largest p_c output that as a prediction
   (b) Discard any remaning (non-max) box with $IOU \geq 0.5$ (associated with the same object so that it overlaps enough with the max) with the box output in the previous step.

Carry out this non-max suppression separately for each object class independently.

Anchor boxes

Each of the grid cells can only detect one object. How do we determine multiple objects in the same grid cell? Say the midpoints of a car and girl are on the same grid cell. We have to pick one of the two object classes.

Define the class label to instead be the two anchor boxes stacked:

$$\mathbf{y} = \begin{bmatrix} p_c & ... & c_3 & p_c & ... & c_3 \end{bmatrix}^T \tag{3.14}$$

Each object in the training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with the highest IOU, in the case of two anchor boxes. Output, $\mathbf{y} \in \mathbb{R}^{3\times3\times16}$. We can use a K-means clustering algorithm to detect multiple shapes of anchor boxes in an image.

3.3.4 Region Proposals

R-CNN: Use color maps to propose regions. Classify the proposed regions one by one, and the output is a label + bounding box. Improvement: Use convolutional implementation of sliding windows to classify all proposed regions.

3.3.5 Semantic Segmentation with U-Net

Goal: Draw a careful outline around the object detected.

Rather than using bounding boxes, the algorithm tries to label each pixel one of the classes of objects. Used by self-driving car teams to determine pixels that represent a drivable road.The size of the output needs to be the same as that of the input, but with a single channel to get the map. To make the image bigger, we use **transpose convolutions**.

Transpose Convolution - learnable upsampling

- The size of the filter is bigger than the input tensor.

- Take the filter and copy ('plop') it over to the output. Input pixels are weights for the filter (we take the dot product of each pixel in the input feature map with the learnable filter). Then shift the window by an amount, 'stride'.

- Sum where output overlaps at an entry. This does NOT compute the inverse of convolution.

- "The transpose convolution forward pass is equivalent to a backward pass of convolution." - Andrej Kaparthy

Overall U-Net Architecture

Use normal convolutions for the first half of the network: shrink the height and width of the image. In the second half of the architecture, use transpose convolutions (deconvolutions) to blow up the height and width, back to the size of the original input image.

Use residual (skip) connections from the earlier layers to the deeper layers of the same size. This is so that spatial information that is more detailed is passed on to deeper layers. The layer right before the second to last layer contains more higher level, lower dimensional (resolution) contextual information. The earlier layer skipped over to the second to last layer has lower level, but higher dimensional information stored. The skip connections from lower layers clears up the boundaries as lower layers have more fine grain details since they have a smaller receptive field.

The output is an $h \times w \times n_{classes}$ tensor, that is a feature map of probabilities that each pixel is a particular class (for that specific layer). Then take the argmax over all the class layers.

Instance Segmentation: Detect instances, give category, label pixels. Given an input image, output all instances of those classes, and for each instance, we want to single out the pixels for that instance. Distinguish between different people, different cows.

SDS - Simultaneous Detection and Segmentation (Instance-aware Semantic Segmentation)

- *Does not use up-sampling and down-sampling (U-Nets), but uses similar pipelines to object detection*
- R-CNNs: offline-external region proposals: it predicts where objects in the image might be located.
- Instance Segmentation is similar to R-CNN but with segments.
- Extract features from a bounding box around the proposed region.
- Mask out background with mean image.
- Cascades: We supervise different intermediate layers as well (at the Region proposal network (RPN), mask instance, and masking out + categorizing instances stages).

Attention Models

RNN only looks at the whole image at once.

3.3.6 Questions / Doubts

- Receptive fields?

- Superpixels and segmentation trees in multi-scale architechtures for semantic segmentation.
- Pose estimation
- Recurrent convolutional network - same CNN weights.

3.4 Week 4

3.4.1 Face Recognition

Face recognition and liveness detection.

- Face verification: input image (name / ID). Output whether the input image is that of the claimed person. (only a binary classifier, yes/no)
- Face recognition: Has a database of K persons. Get an input image. Output ID if the image is any of the K persons (or not recognized). Need a 99.9 % for 100 person database.

3.4.2 One-shot Learning Problem

What do we do if we only have 1 example of a person? We have to learn from just one example to recognize the person again.

Learning a *similarity function*:

$$d(img1, img2) = \text{degree of difference between images} \tag{3.15}$$

$$\text{if } d(img1, img2) \leq \tau \text{ same person} \tag{3.16}$$

$$\text{if } d(img1, img2) > \tau \text{ different person} \tag{3.17}$$

The new image $img2$ is compared to all existing faces in the data base (using pairwise comparisons). So how do we train a neural network to learn this function d?

3.4.3 Siamese Network

$f(\mathbf{x}^{(1)})$ is an encoding of $x^{(1)}$.

We can feed the faces of other people into the same neural convolutional network to encode the second, third, ... inputs.

The output layers are 128×1 column vectors, and we *do not* use softmax there. We always use the same learned parameters.

$$d(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}) = ||f(\mathbf{x}^{(1)}) - f(\mathbf{x}^{(2)})||^2 \tag{3.18}$$

- Parameters of the NN define an encoding f
- learn parameters so that if two examples are of the same person, d is small. If two examples are of different people, d is large. Use backprop to learn the parameters.

3.4.4 Objective Function Defined: Triplet Loss

Learn the parameters of the NN, we have to look at the encodings of multiple images. Always look at an anchor image. We want the d to be small when compared with a positive example. We look at three images at a time, an anchor (A), positive (P), and a negative (N).

$$\text{Want: } ||f(A) - f(P)||^2 + \alpha \leq ||f(A) - f(N)||^2$$
$$\implies ||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha \leq 0$$

One trivial way to make this condition hold through learning is to learn the encoding $f(img) = \mathbf{0}$. **Solution:** To make sure not all encodings are equal to each other for all images, the α (margin parameter) is added.

Loss function

Given 3 images, A, P, N (N is of a different person compared to the other 2),

$$\mathcal{L}(A, P, N) = max\left(||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha, 0\right) \tag{3.19}$$

If the difference is negative, then the loss is zero. This is what we want (see the green box above). Otherwise we have a positive loss that we want to minimize.

$$J = \sum_{i=1}^{m} \mathcal{L}\left(A^{(i)}, P^{(i)}, N^{(i)}\right) \tag{3.20}$$

We need a *training* dataset of atleast 2 pictures of the same person. The one-shot algorithm only works when testing. Advice: choose triplets that are hard to train on, not randomly (where the condition in the green box is not satisfied).

Binary Classification Alternative to the Triplet Loss

This is also a binary classification problem.

Take the two encodings of two examples and take the logistic (sigmoid) activation on the final output layer to output $\hat{y}$.

$$\hat{y} = \sigma\left(\sum_{k=1}^{128} w_k |f(x^{(i)})_k - f(x^{(j)})_k| + b\right) \tag{3.21}$$

Use precomputing of the encodings of employees.

Supervised Learning is another alternative

3.4.5 Neural Style Transfer

We can generate a new image drawn in a certain style over the content image. The output is a generated image.

We pick a unit in layer 1 (hidden). Find the 9 image patches that maximize the unit's activation. Repeat for other 9 units.

A deeper layer unit has a much larger receptive field. The deeper layers are detecting patterns that are more complex and higher level.

3.4.6 Cost Function [Gatys et al., 2015]

$$J(G) = \alpha J_{content}(C, G) + \beta J_{style}(S, G) \tag{3.22}$$

Find the generated image G:

1. Initiate G randomly: $100 \times 100 \times 3$

2. Use gradient descent to minimize $J(G)$. Update rule (updating the pixels of the output):

$$G := G - \frac{\partial}{\partial G} J(G) \tag{3.23}$$

Content Cost Function

Say you use hidden layer l to compute content cost. Use pre-trained ConvNet (VGG)

Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on the images. If these are similar, both images have similar content. Use this element wise difference between hidden layer activation unrolled into vectors. This similarity should hold (minimized J, or difference metric) for some intermediate layer only so that G isn't identical to C.

$$J_{content}(C, G) = \frac{1}{2} || a^{[l](C)} - a^{[l](G)} ||^2 \tag{3.24}$$

Style Cost Function

Assume we are using layer l's activation to measure style.

Define style as correlation between activations across channels.

- How correlated are the activations accross different channels? How often do certain high level features occur?

(rewatch)

Let $a^{[l]}_{i,j,k}$ = activation at (i, j, k).

$G^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$ square matrix

$G^{[l]}_{kk'}$ is the measure of correlation between an activation in channel k and activation in channel k'. Here, $k \in 1...n_c^{[l]}$.Unnormalized cross-covariance:

$$G^{[l]}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l]}_{ijk} a^{[l]}_{ijk'} \tag{3.25}$$

We compute this G value for both the style and the generated image. These are called the gram matrices in linear algebra.

$$J^{[l]}_{style}(S, G) = \frac{1}{\dots} ||G^{[l](S)} - G^{[l](G)}||^2_F \tag{3.26}$$

$$J^{[l]}_{style}(S, G) = \frac{1}{2n_H^{[l]} n_W^{[l]} n_C^{[l]}} \sum_k \sum_{k'} (G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'}) \tag{3.27}$$

Style cost function from multiple layers:

$$" = \sum_l \lambda^{[l]} J^{[l]}_{style}(S, G) \tag{3.28}$$

(Frobenius norm)

We can loop over all the other layers too.

3.4.7 Questions / Doubts

No, Neural style transfer is about training on the pixels of an image to make it look artistic, it is not learning any parameters.