# Common Considerations

## Preprocessing

* + 1. Zero-center
       1. In general, we always subtract the mean across every feature (X -= np.mean(X, axis=0))
       2. To zero-center images, we usually subtract the mean over all pixels (X -= np.mean(X)) or separately across three color channels (X -= np.mean(X, axis=(0, 1, 2)))
       3. Why do we want zero-mean data as the input to neural network?
          1. When the input to a neuron is always positive, the gradients on weights w are all positive or all negative. In fact, any sort of bias could cause such problems
    2. Normalization

Apply normalization if the input features have different scales (or units), but they should be of approximately equal importance to the learning algorithm

* + - 1. Divide each feature by its standard deviation once it has been zero-centered
      2. Normalize each feature so that the min and max along the feature is -1 and 1
    1. In computer vision, we always perform zero-center and standard normalization but don’t do further preprocessing, such as PCA, and whitening, since pixels generally have relatively comparable scaled distribution
    2. Why do we perform data normalization?
       1. If we don’t do this, some features will be weighted more in the loss function and some will be weighted less. Data normalization makes all feature weighted equally

## Initialization

* + 1. It is in general recommended to initialize the weights using  
       w = np.random.randn(in, out) / np.sqrt(in)  
       Which is called the Xavier initialization.  
       For ReLU neurons, [this paper](https://arxiv.org/abs/1502.01852) recommends to initialize weights using  
       w = np.random.randn(in, out) / np.sqrt(in/2)  
       Which is called the Kaiming initialization

sqrt(in) and sqrt(in/2) above calibrate the variances of the outputs — The distribution of the outputs from simple randomly initialized weights has a variance that grows with the number of inputs

* + - 1. The reason to divide the random initialization by np.sqrt(n) in Xavier initialization is to calibrate the variance of the outputs so that the outputs have the same variance as the inputs. Let the outputs s = wx, the variance is computed  
          The fourth step is derived based on the assumption of zero mean inputs and weights — that is, E[w(i)]=E[x(i)]=0. To make sure Var(s)=Var(x), Var(w) should be 1/n — that explains why the sample weights from the normal distribution(whose variance = 1) is divided by the square root of n

n is the number of inputs

* + - 1. /2 introduced for ReLU units is due to the observation that ReLU kills half of inputs (the negative half, assuming the inputs have zero mean)
    1. It is common to simply initialize the biases to be zero
    2. Why not initialize all weights to zero?
       1. The same initialization suggests the same updates on weights. That means there is no difference between all neurons.
    3. Deep networks are more sensitive to the initialization weights,

## Activation function

A type of functions used to break linearity

* + 1. Why do neural networks need activation functions?
       1. To break linearity. If we don’t use any activation function, it’ll be meaningless to build a multi-layer network, since a multi-layer network can be reduced to one linear node in that case.

For a simplest example, we define a network with only one input variable and one hidden layer, and without activation function. It has w1, b1 as its weights and bias at input layer, w2, b2 at hidden layer. This network can be reduced to a single linear node which have w2(w1x+b1)+b2 as it’s linear function.

* + - 1. Activation helps squash the unbounded linearly weighted sum into a certain range so as to avoid large values accumulating high up the processing hierarchy
      2. It’s differentiable, which makes them qualified for gradient descent
    1. What kind of requirements must functions satisfy in order be used as activation functions?
       1. Activation functions must be continuous and differentiable so that the model could perform backprop latter
    2. Sigmoid function

S(x) = 1/(1+e^(-x)). A wide variety of sigmoid functions have been used as the activation function of artificial neurons

Sigmoid function has a very nice derivative:

* + - 1. def sigmoid(x):  
          # do not use np.exp(x), it will overflow for large x  
          return 1/(1+np.exp(-x))  
         def sigmoid\_prime(x):  
          return sigmoid(x)\*(1-sigmoid(x))
      2. To achieve numerically stable, we need to treat x≥0 and x <0 separately
      3. What’s the drawback of using sigmoid function as the activation function?
         1. The vanishing gradient problem: sigmoids saturate and kill gradients
         2. Sigmoid outputs are not zero-centered: if the input to a neural is always positive, then the local gradient on the weights w will become either all positive or all negative during backpropagation. This will introduce undesirable zig-zagging dynamic in the gradient updates for the weights.
    1. Softmax function

* + - 1. def softmax(Z):  
          expZ = np.exp(Z)  
          expSum = sum(expZ)  
          return [ele/expSum for ele in expZ]
      2. Softmax suffers numeric instability, how do we get rid of such a issue?
         1. Instability caused by exponential to large z, we can get rid of it by subtracting zs by the greatest z.   
            Letting  
             now we have
      3. Define loss function  
          And softmax  
          The partial derivative of L w.r.t. z\_k is
    1. tanh (hyperbolic tangent function)

A better choice which has range in (-1, 1), but still suffers the vanishing gradient problem

Its derivative is

* + 1. ReLU (rectified linear unit)

, a good choice for hidden layer which helps with the vanishing gradient problem

* + - 1. Advantage:  
         1. It was found to greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid/tanh functions  
         2. It doesn’t involve expensive operation compared to the sigmoid/tanh functions which requires exponentials
      2. Disadvantage:  
         ReLU units can be fragile during training and can “die”. A dead ReLU always outputs the same value (zero as it happens) for any inputs, possibly caused by a large negative bias term.

Although such dead units is likely to stay dead, there’s still a chance that they get revived by the change of inputs.

With a proper setting of the learning rate, this is less frequently an issue

* + 1. Leaky ReLU

, where a < 1, a usual choice for GAN which attempts to fix the “dying ReLU” problem

* + 1. Maxout

, a generalization of ReLU and leaky ReLU (for example, for ReLU we have w1, b1 = 0). Maxout has all the benefits of the ReLU and does not have its drawbacks. However, it doubles the number of parameters for each neuron, leading to a high total number of parameters

## Error function / Loss function / Cost function

A function used as a measurement that tells us how far apart the current model is from the goal or how bad the input is.

Heads up, this is where the optimization begins

* + 1. How can we tell whether one model is better than another? For example, for single-layer feedforward neural networks with different weights and using sigmoid as their activation functions, how can we tell which perceptron is the best model?
       1. The method we use is called maximum likelihood estimation(MLE). In the above example, the model with the weights, [W, b], maximizing the likelihood of predictions being in accord with observations is the best one.

For a concrete example, assume we have two models: M1 with [W1, b1] as its weights, M2 with [W2, b2] as its weights. We also have two points x1, y as data

Observation: x1 is labeled 1, y is labeled 0

Prediction: P1 outputs 0.8 for x1 and 0.4 for x2, P2 outputs 0.7 for x1 and 0.3 for x2

The probabilities the perceptron returns suggest the likelihood (the algorithm predicts) that points are labeled 1. Thus, P1 predicts the probability of x2 being labeled 0 is 1-0.4, which is 0.6, and P2 predicts the probability of y being labeled 0 is 1-0.3, which is 0.7. Now, the likelihood of predictions being in accord with observations for P1 is 0.8\*0.6, which is 0.48, and that for P2 is 0.7\*0.7, which is 0.49. Since 0.48<0.49, P2 is better than P1

* + - 1. In the above process, we compare models through the likelihood, which is obtained through multiplication, but multiplication gets harder when the data gets more and more. Is there another way that simplifies the process without affecting the results?
         1. We can apply log to the multiplication based on the observation that log(ab)=log(a)+log(b). Furthermore, if we take the negative of the logarithm, we’ll end up something that is small for good model and great for bad model. That’s cross-entropy
    1. Classification
       1. Cross-entropy

The sum of the negatives of the logarithms of the probabilities of predictions being in accord with observations (always divided by the number of inputs)

* + - * 1. In the above definition of cross-entropy, we use the probabilities of predictions being in accord with observation as if they’re given. What if we don’t have such probabilities—the case where we only know the observations (y, either 0 or 1) and the probabilities (f) of points being labeled 1, is there any general formula for that?

def cross\_entropy(y, f):  
 assert len(y) == len(f)  
 # Convert to numpy array so as to perform arithmetic operations on array  
 y = np.array(y)  
 f = np.array(f)  
 return -np.sum(y \* np.log(f) + (1 - y) \* np.log(1 - f))

* + - * 1. Categorical cross-entropy

A variant of cross-entropy dealing with the case where the observations is more than two classes

Why softmax? Why not use something like ?

We need to make sure that   
1. every probability is nonnegative   
2. all of probabilities add up to 1

Now that we have one hot encoded observations and the corresponding probabilities, we are good enough to define categorical cross-entropy:

For example:

we have the probability table

For the observation that there is a Beaver behind Door1, a Walrus behind Door2, and a Duck behind Door3, we have

Y = [[0, 1, 0],

[0, 0, 1],

[1, 0, 0]]

* + - * 1. Why do we bother to use cross-entropy to compare models?

Based on the fact that right-placed points often have higher probabilities of predictions being in accord with observations, which means the corresponding cross-entropy is smaller. That makes it a good practice to use cross-entropy as a loss function

* + - * 1. Information theory view

The cross-entropy between a “true” probability p and an estimated distribution q is defined as

1/q(x) is the frequency of x. log returns the bits required to represent x

* + - 1. Hinge loss / SVM loss
         1. For binary classification

y is either 1 or -1

* + - * 1. For multi-class classification

If the logit for an incorrect category is greater than that for the correct category by some margin (here, it’s 1) we add it to loss to penalize it.

* + 1. Attribute classification

In which the label y is a binary vector where every element indicates whether or not have a certain attribute, and where the attributes are not exclusive

* + - 1. One sensible approach in this case is to define the loss function using SVM loss  
          Where the inner sum is over all categories j, y(i, j) is either +1 or -1 depending on whether the example is labeled with the j-th attribute, and the score vector f will be positive when the class is predicted to be present and negative otherwise

The loss is accumulated if a positive example has score less than +1, or when a negative example has score greater than -1

* + - 1. An alternative is to train a logistic classifier for every attribute independently  
          Where the inner sum is the logistic loss over all categories j, y(i, j) is either 1 (positive) or 0 (negative)
    1. Regression
       1. Least squares

The method of least squares is a standard approach in regression analysis to the approximate solution of overdetermined system, i.e., sets of equations in which there are more equations than unknowns, or for real problems, there are more samples than independent variables. "Least squares" means that the overall solution minimizes the sum of the squares of the residuals (a residual being: the difference between an observed value, and the fitted value provided by a model) made in the results of every single equation.

* + - * 1. Limitation:  
           Simple regression and least-squares methods have problems when the problem has substantial uncertainties in the independent variables (aka, regressors), that is, there are measurement errors in the independent variables. In such cases, the methodology required for fitting [errors-in-variables models](https://en.wikipedia.org/wiki/Errors-in-variables_models) may be considered instead of that for least squares.
      1. The regression loss is much more fragile and harder to optimize than a classification loss, because it requires the network to output exactly one correct value for each input and it can introduce huge gradient (least squares). Applying dropout in the network with the L2 loss is not a great idea (especially in the layer right before the L2 loss)

Always consider classification first (e.g. to quantize the output into bins), apply regression only when necessary.

## Regularization

Any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error

* + 1. L1 regularization: add 𝛌(|w1| + |w2|…)  
       L2 regularization(weight decay): add 𝛌(w1^2 + w2^2 + …)

We don’t add penalty to biases, since they don’t control the strength of influence of an input dimension

* + 1. How does regularization help with overfitting?
       1. Keeping weights small helps do gradient descent on activation functions such as tanh, sigmoid, etc. Furthermore, when x is small, such activation functions are nearly linear, and thus help simplify the network
       2. It makes some weights close to zero and thereby takes the same effect as dropout
    2. What’s the difference between L1 and L2 regularization? How to decide between them?
       1. L1 regularization results in vectors with sparse weights; that means, small weights will tend to be 0. It also helps for future selection, it tells which features are important and turn the other to 0  
          L2 regularization produces vectors with small homogeneous weights; that is, it doesn’t have obviously preference for any feature. It normally gives better results for training models
       2. If our aim is to find the dominate features, L1 is preferred  
          If the aim is to train the model, L2 is preferred
    3. Add random noise at training time, marginalize over the noise at testing time
    4. Batch Normalization

The idea that normalizes inputs to nonlinearities

* + - 1. For an input value x{i}, it is normalized to  
          Then, it’s further transformed to

mu is the mean, sigma^2 is the variance, epsilon, can be any small positive constant. Here we add epsilon because

1. The global variance is higher than the mini-batch variance used here — recall that the global variance is n times the mini-batch variance

2. Doing so helps maintain numeric stability: avoid blowing up caused by small variance

gamma, and beta, are learnable parameters of the network and serve to scale and shift the normalized value, respectively

* + - 1. Batch normalization uses weights as usual, but does not add a bias term. This is because its calculations include gamma and beta variables that make the bias term unnecessary.
      2. Why do we introduce batch normalization?
         1. The training is complicated by the fact that the inputs to each layer are affected by the parameters of all preceding layers — so that small changes to the network parameters amplify as the network becomes deeper, which leads to internal covariate shift

The “internal” here refers to the change in the distribution of internal neurons

* + - * 1. Covariate shift

The input distribution to a learning system changes but the conditional distribution of the output doesn’t

* + - * 1. An illustration shows how batch normalization helps optimization

The left figure shows the data distribution before batch normalization, and the right shows that after batch normalization.

Left figure: the classification loss is very sensitive to small perturbations in weights, which makes it hard to optimize

Right figure: the classification is less sensitive to small changes in parameter values, which makes it easier to optimize

* + - 1. Benefits of batch normalization
         1. Networks train faster – Each training iteration will actually be slower because of the extra calculations during the forward pass and the additional parameters to train during back propagation. However, it should converge much more quickly, so training should be faster overall.
         2. Allows higher learning rates – Gradient descent usually requires small learning rates for the network to converge. And as networks get deeper, their gradients get smaller during back propagation so they require even more iterations. Using batch normalization allows us to use much higher learning rates, which further increases the speed at which networks train.
         3. Makes weights easier to initialize – Weight initialization can be difficult, and it's even more difficult when creating deeper networks. Batch normalization seems to allow us to be much less careful about choosing our initial starting weights.
         4. Makes more activation functions viable – Some activation functions do not work well in some situations. Sigmoids lose their gradient pretty quickly, which means they can't be used in deep networks. And ReLUs often die out during training, where they stop learning completely, so we need to be careful about the range of values fed into them. Because batch normalization regulates the values going into each activation function, non-linearilities that don't seem to work well in deep networks actually become viable again.
         5. Simplifies the creation of deeper networks – Because of the first four items listed above, it is easier to build and faster to train deeper neural networks when using batch normalization. And it's been shown that deeper networks generally produce better results, so that's great.
         6. Provides a bit of regularization – Batch normalization adds a little noise to your network. In some cases, such as in Inception modules, batch normalization has been shown to work as well as dropout. But in general, consider batch normalization as a bit of extra regularization, possibly allowing you to reduce some of the dropout you might add to a network.
         7. May give better results overall – Some tests seem to show batch normalization actually improves the training results. However, it's really an optimization to help train faster, so you shouldn't think of it as a way to make your network better. But since it lets you train networks faster, that means you can iterate over more designs more quickly. It also lets you build deeper networks, which are usually better. So when you factor in everything, you're probably going to end up with better results if you build your networks with batch normalization.
      2. Where do we use batch normalization?
         1. Don’t use it in the first and last layer, i.e., the input and output layer

For a discriminator, having batch normalization in the input layer makes it hart to make sure the pixels coming out of the generator have the correct mean and standard deviation

* + - * 1. Apply it before activation function
        2. Don’t use batch normalization in the layer of the discriminator from which we want to extract the different average value of features for different input data so that we can calculate the [extra term in the loss function for the generator](omnioutliner:///open?focus=lmYu3eIo0TY&row=aDlp7-IgeWn)

If we do, the average value of features will be almost the same for any data (no matter it’s training data or generated data), which is equal to \beta . The reasoning is shown as below

The last step holds because \mu is the mean of all x{i}

* + - 1. How to add batch normalization into networks?
         1. High level

Define a Boolean placeholder training, which will be passed to batch\_normalization to indicate whether it is training or inferring

Add tf.layers.batch\_normalization before activation functions

Define optimization as below  
with tf.control\_dependencies(tf.get\_collection(tf.GraphKeys.UPDATE\_OPS)):  
 train\_opt = tf.train.AdamOptimizer(learning\_rate).minimize(loss)

For training steps, set training True; for testing steps, set it False

* + - * 1. Low level

Define a Boolean placeholder training as before

Define variables used in normalization  
gamma = tf.Variable(tf.ones([num\_units]))  
beta = tf.Variable(tf.zeros([num\_units]))  
mean = tf.Variable(tf.zeros([num\_units]), trainable=False)  
variance = tf.Variable(tf.ones([num\_units]), trainable=False)  
epsilon = 1e-3

Define batch\_normalization functions for training and testing, respectively  
def batch\_training():  
 batch\_mean, batch\_variance = tf.nn.moments(layer, [0])  
   
 decay = .99  
 train\_mean = tf.assign(mean, mean \* decay + batch\_mean \* (1-decay))  
 train\_variance = tf.assign(variance, variance \* decay + batch\_variance \* (1-decay))  
   
 with tf.control\_dependencies([train\_mean, train\_variance]):  
 # notice here we pass batch\_mean and batch\_variance as arguments  
 return tf.nn.batch\_normalization(layer, batch\_mean, batch\_variance, beta, gamma, epsilon)  
   
def batch\_testing():  
 return tf.nn.batch\_normalization(layer, mean, variance, beta, gamma, epsilon)  
layer = tf.cond(training, batch\_training, batch\_testing)

For training steps, set training True; for testing steps, set it false

* + - 1. Batch normalized CNNs
         1. To sustain the convolutional property, the mean and variance are computed over all locations in a mini-batch of feature maps
         2. Batch normalization for ConvNet can be simply implemented based on the ordinary version

def spatial\_batchnorm\_forward(X, gamma, beta, bn\_param):  
 N, C, H, W = X.shape  
 X = X.transpose(0, 2, 3, 1).reshape(-1, C)  
 out, cache = ordinary\_batchnorm\_forward(X, gamma, beta, bn\_param)  
 out = out.reshape(N, H, W, C).transpose(0, 3, 1, 2)  
 return out, cache

def spatial\_batchnorm\_backward(dout, cache):  
 N, C, H, W = dout  
 dout = dout.transpose(0, 2, 3, 1).reshape(-1, C)  
 dx, dgamma, dbeta = ordinary\_batchnorm\_backward(dout, cache)  
 dout = dout.reshape(N, H, W, C).transpose(0, 3, 1, 2)  
 return dx, dgamma, dbeta

* + - 1. Readings
         1. There are good mathematical reasons why batch normalization helps the network learn better. It helps combat what the authors call internal covariate shift. This discussion is best handled [in the paper](https://arxiv.org/pdf/1502.03167.pdf) and in [Deep Learning](http://www.deeplearningbook.org/) a book. Specifically, check out the batch normalization section of [Chapter 8: Optimization for Training Deep Models](http://www.deeplearningbook.org/contents/optimization.html).
    1. Layer Normalization
       1. Layer normalization uses mean and variance over the features instead over sample space. Thereby it mitigates the problem that batch normalization computes mean and variance over the batch, which is restricted by hardware.
       2. Layer normalization doesn’t perform as well as batch normalization when used with convolutional layer
    2. Group Normalization
       1. Group Normalization works effectively with convolutional net
    3. Dropout

A method to prevent overfitting: At each epoch, some neural units are dropped out of the network with probability p

* + - 1. Inverted dropout
         1. When implementing dropout, we often use a boolean matrix to do the job:  
             Dividing the dropout matrix by p is to make sure that the expected values of the outputs remains the same, since dropout roughly rescales the expected outputs by p
      2. Why does dropout help with overfitting?
         1. Prevent co-adaptation of features
         2. Dropout is training a large ensemble of models
         3. By shutting down some nodes, it diminishes the number of units in the network
      3. How do we decide how much keep-probability for a layer when applying dropout?
         1. Layers with more hidden units tends to overfitting, so we give lower keep-probabilities to those layers
      4. One downside of dropout is that using dropout makes the loss function less well defined, since dropout randomly kills some units at each iteration. That may result in fluctuations in loss plot.
      5. Don’t use dropout at test time. It’s a way to prevent overfitting, using dropout at test time only introduces noise to predictions

## Optimization

* + 1. First order methods
       1. Gradient descent

An first-order iterative optimization algorithm for finding the minimum of a function

* + - * 1. What’s the gradient of the error function—sigmoid cross-entropy?

The gradient of E, at a point x = (x{1}, …, x{n}) is   
 For the case where the activation function is the sigmoid function, after a series of calculations, we finally obtain

* + - * 1. How to minimize the error function?

For every point x = (x{1}, …, x{n}), we update the weights (W, b) to (W, b) - a∇E

a is learning rate, ∇E is the gradient of E.

* + - * 1. Implementation

def prediction(Xs, W, b):  
 """ calculate the prediction """  
 return sigmoid(np.matmul(Xs, W) + b)  
  
def delta\_error(Xs, ys, y\_hats):  
 """ calculate the gradients with respect to W and b """  
 # the gradients with respect to W  
 gradients\_w = [np.mean([Xs[i][j] \* -(ys[i] - y\_hats[i]) for i in range(len(ys))]) for j in range(len(Xs[0]))]  
 # the gradient with respect to b  
 gradient\_b = np.mean([-(ys[i] - y\_hats[i]) for i in range(len(ys))])  
 return gradients\_w, gradient\_b  
  
def gradientDescentStep(Xs, ys, W, b, learn\_rate=0.01):  
 """ apply gradient descent to update W and b """  
 y\_hats = prediction(Xs, W, b)  
 gradients\_w, gradient\_b = delta\_error(Xs, ys, y\_hats)  
 # update the weights  
 W -= np.array([learn\_rate \* gw for gw in gradients\_w])  
 b -= learn\_rate \* gradient\_b  
 return W, b

* + - * 1. Stochastic gradient descent

Divide input data into several batches, each batch goes through the neural network as before at each step. Each step is less accurate than the one that takes all data at one time, in practice, however, it’s much better to take a bunch of slightly inaccurate steps than to take a good one

* + - * 1. How to get out of the local minima when we do gradient descent?

Random start

Start with different random weights, and do gradient descent for all of them

* + - * 1. Backpropagation

A method to to calculate the error contribution of each neuron after a batch of data is processed

* + - * 1. What’s the difference between gradient descent and backpropagation?

Gradient descent is a first-order optimization method, with which the neural network gets trained. Backpropagation is a technique to calculate gradients, which is often used by gradient descent as a gradient-computing technique

* + - * 1. What’s the drawback of SGD?

SGD updates the loss very slowly since the sensitivity of the loss to different weights varies, which results in zig-zag update

SGD gets stuck at a local minimum or saddle point

Saddle point is actually much more common in high dimensional space

Gradients coming from minibatches can be noisy

* + - 1. SGD with momentum
         1. Momentum resorts to the physical concept of velocity to smooth out the gradient descent  
             In some literature, the coefficient 1-beta is omitted. In that case, the velocity is the discounted accumulative gradients (which could be viewed as the acceleration in physics)  
            v(n)=dW(n) + beta dW(n-1) + beta^2 dW(n-2)…  
            But [Andrew Ng in this video](https://www.youtube.com/watch?v=k8fTYJPd3_I) personally recommends to keep 1-beta since omitting it indicates velocity is rescaled when we tune beta, which leads to further tuning alpha if we want the step size to stay in the same scale

A recommended choice for beta is 0.9

* + - 1. SGD with Nesterov momentum
         1. Instead of computing the gradient at the current position W, Nesterov momentum computes the gradient at W’ = W+beta v  
             In practice, to align with the original momentum (which corrects the velocity first and then updates the weights), people would like to use W’ as parameter directly instead of W, then the update rule becomes   
             Since v starts with value 0, W’ is initially equal to W. At convergence of optimization, when v approximates to 0, W’ will be identical to W as well. These observations make W’ a well-defined replacement for W
         2. Nesterov differs from vanilla momentum in that   
            Vanilla momentum first corrects velocity and then make a big step according to that velocity  
            Nesterov first make a step into velocity direction, and then corrects the velocity vector based on the new location and adjusts the location accordingly
      2. Adagrad

Abbr. for adaptive gradient

* + - * 1. Adagrad adjusts the step size so that the weights that receive high gradients will have their step size reduced, while weights that receive small gradients will have their step size increased   
            epsilon is usually set somewhere in range from 1e-4 to 1e-8. The square root operation in the denominator turns out to be very important and without it the algorithm performs much worse
        2. Adagrad accumulates the squared gradients, which gradually slows down the learning process. It is desirable in convex problems but it’s usually undesired in deep learning since it's too aggressive and stops learning too early
      1. RMSprop

Abbr. for root mean square prop

* + - * 1. RMSprop only differs from Adagrad in that it uses a moving average of square gradients instead  
            In this way, it still modulates the step size of each weight based on the magnitudes of its gradients, but doesn’t monotonically slow down the learning process

rho is typically set to 0.9, 0.99, 0.999

* + - 1. Adam

Abbr. for adaptive moment estimation

* + - * 1. Adam combines the ideas from both momentum and RMSprop  
            m1 is equal to the velocity in the momentum, m2 is equal to the cache variable in the RMSprop.  
           Bias correction compensates for the fact that m1 and m2 are biased at zero in the first few time steps, and thereby leads to large step size at the beginning, which may even not be invertible in some cases.

beta1 = 0.9

beta2 - 0.999

epsilon = 1e-8

t in the bias correction step is the iteration number

* + - 1. Adam is currently recommended as the default algorithm to use
      2. Most optimization algorithm assume that the data are iid. i.e. independently and identically distributed.

## Model ensemble

* + 1. Same model, same hyperparameters, different initializations
    2. Same model, different hyperparameters, same initializations
    3. Different checkpoints of a single model
    4. An extra network which maintains exponentially decaying sum of previous weights during training

## Hyperparameters

* + 1. Learning rate annealing
       1. Step decay

which is slightly preferable in practice since the hyperparameters it involves (the fraction of decay and the step timings in units of epochs)are more interpretable than the hyperparameter k

* + - * 1. The learning rate is reduced by a half every 5 epochs or by 0.1 every 20 epochs. These numbers depend heavily on the type of problem and the model.

A heuristic is to reduce the learning rate when the validation error stops improving

* + - 1. Exponential decay
         1. a=a(0)e^{-kt}

a(0) is the initial learning rate

k is the hyperparameter,

t is the iteration number

* + - 1. 1/t decay
         1. a=a(0)/(1+kt)
      2. Learning rate annealing is generally work with SGD+momentum, but not with algorithms such as Adam since these algorithms have already encompassed some adaptive learning rate methods
      3. A good starting point of learning rate is 1e-3, 5e-4
    1. Minibatch size
       1. Large minibatch sizes gain computational boosts that utilizes matrix multiplications in the training calculations. That comes at the expense of more memory and computational resources.
       2. Large minibatch sizes may result in worse accuracy
       3. Small minibatch sizes has more noise in the error calculations, this help to escape from the local minimum
       4. 32, 64, 128, 256 are good choice to starting values
    2. Epoch
       1. Implement early stopping using [Hooks](https://www.tensorflow.org/api_guides/python/train#Training_Hooks), which allow to stop the training when there is no improvements
    3. Hidden units

The number of hidden units is a direct measurement of learning capacity

* + - 1. A heuristic for the first hidden layer is that setting it to a number larger than the number of inputs
      2. Generally speaking, 3-layer networks outperform two-layer ones, but going even deeper rarely helps much more. CNNs are an exception to this, since the deeper CNN are, the better they perform
      3. Too many hidden units leads to overfitting, for it just tries to memorize the training set.
    1. Further readings
       1. If you want to learn more about hyperparameters, these are some great resources on the topic:  
           • [Practical recommendations for gradient-based training of deep architectures](https://arxiv.org/abs/1206.5533) by Yoshua Bengio  
           • [Deep Learning book - chapter 11.4: Selecting Hyperparameters](http://www.deeplearningbook.org/contents/guidelines.html) by Ian Goodfellow, Yoshua Bengio, Aaron Courville  
           • [Neural Networks and Deep Learning book - Chapter 3: How to choose a neural network's hyper-parameters?](http://neuralnetworksanddeeplearning.com/chap3.html#how_to_choose_a_neural_network's_hyper-parameters) by Michael Nielsen  
           • [Efficient BackProp (pdf)](http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf) by Yann LeCun  
          More specialized sources:  
           • [How to Generate a Good Word Embedding?](https://arxiv.org/abs/1507.05523) by Siwei Lai, Kang Liu, Liheng Xu, Jun Zhao  
           • [Systematic evaluation of CNN advances on the ImageNet](https://arxiv.org/abs/1606.02228) by Dmytro Mishkin, Nikolay Sergievskiy, Jiri Matas  
           • [Visualizing and Understanding Recurrent Networks](https://arxiv.org/abs/1506.02078) by Andrej Karpathy, Justin Johnson, Li Fei-Fei
    2. Embedding

Which translate large sparse vectors into a lower-dimensional space that preserves semantic relationships

* + - 1. Sparse representations such as one-hot encoding have a couple of problems that can make it hard for a model to learn effectively
         1. Huge input vectors (M) means a super-large number (MN) of weights. This lead to  
            - More training data  
            - More computation

M is the dimension of input layer

N is the dimension of the first hidden layer

* + - * 1. Similar inputs share no meaningful relations with each other
      1. Position (distance and direction) in the vector space can encode semantics in a good embedding
      2. A recommended initial trial for the size of embedding is
      3. Notice that XW, where X is one-hot encoded whose ith element is 1, results in the ith column in W. In light of this, we could use the ith column in W to represent X, which is exactly what embedding does. In fact, we don’t need to represent X in one hot encoding form, X is fine just to be an integer, i in above example.
      4. For a bag of inputs, we could retrieve the embedding for each individual item and then add them together
      5. To obtain an appropriate W, we create a one-layer network, and use as (inputs, output) pairs (X, Y), where X is a word in a sentence and Y is a word picked in X’s vicinity, to train the network

Use tf.nn.sampled\_softmax\_loss for loss calculation so as to gain some speed-up

* + - * 1. n\_vocab = 1e5 # Number of words in vocabulary   
           n\_embedding = 200 # Number of embedding features   
           n\_sampled = 100 # Number of negative labels to sample  
           train\_graph = tf.Graph()  
           with train\_graph.as\_default():  
            # Placeholders  
            inputs = tf.placeholder(tf.int32, (None), name='inputs')  
            # Labels is required to have a shape of (batch\_size, num\_true) by tf.nn.sampled\_softmax\_loss  
            labels = tf.placeholder(tf.int32, (None, None), name='labels')  
            # Embedding layer  
            embedding = tf.Variable(tf.random\_uniform((n\_vocab, n\_embedding), -1, 1)) # create embedding weight matrix here  
            embed = tf.nn.embedding\_lookup(embedding, inputs)# use tf.nn.embedding\_lookup to get the hidden layer output  
            # Output layer  
            softmax\_w = tf.Variable(tf.truncated\_normal((n\_vocab, n\_embedding), stddev=.1))  
            softmax\_b = tf.Variable(tf.zeros(n\_vocab))  
              
            # Calculate the loss using negative sampling  
            loss = tf.nn.sampled\_softmax\_loss(weights=softmax\_w, biases=softmax\_b,   
            labels=labels, inputs=embed,  
            num\_sampled=n\_sampled, num\_classes=n\_vocab)  
              
            cost = tf.reduce\_mean(loss)  
            optimizer = tf.train.AdamOptimizer().minimize(cost)
    1. Lambda, C in loss

Which controls the tradeoff between the data loss and the regularization loss

* + - 1. Small weights result in lower regularization loss, they may also cause the probability out of the softmax diffuse and thereby results in high data loss.  
         As an example, suppose that the values for some three classes come out to be [1, -2, 0], the softmax function would then compute  
          Now suppose the weights became half smaller because of regularization, the values in turns becomes [0.5, -1, 0], the softmax function would then compute
    1. Why do we bother to define a validation set other than a testing set?
       1. The validation set provides a way to indicate what’s the best hyperparameters for the model (wherein we may have bias to choose the hyperparameters that best fits the validation set), while on the other hand, the testing set actually tells us how the algorithm is doing on the unseen data

The testing set is the dataset which we should touch at the end of the day, and we should not tune our algorithm based on that.

## Precision and recall

* + 1. For skewed classes, simple error metrics such as accuracy or loss may not work well as expected. For example, a learning algorithm that predicts whether a patient has cancer achieves 99% accuracy. However, the actual possibility that a patient has cancer is just 0.1%. This leads to a simple algorithm that predicts no cancer all the time achieving 99.9% accuracy which is higher than the pervious learning algorithm.
    2. To better evaluate a learning algorithm when the class is skewed, we resort to the concepts of precision and recall. Given a confusion matrix  
        Precision is defined to be the ratio of true positive to prediction positive (true positive plus false positive)  
       Recall is defined to be the ratio of true positive to condition positive (true positive plus false negative)

Generally, we define the rare class as positive when we talk about precision and recall

* + - 1. High precision results from high threshold while high recall from low threshold
      2. F1 score (F score): 2PR/(P+R)
    1. Average precision

## When learning algorithms go south

* + 1. Check the input data
       1. If there are some obvious noisy samples, get rid of them
       2. Extract more features, create some related, drop some irrelevant.
       3. Normalize / standardize the data if some features vary in a range obviously larger than the others do
          1. sklearn.preprocessing.scale
       4. Generate more data
          1. Data augmentation
       5. Change perspective
          1. Embedding
          2. Gaussian kernel in SVM
          3. sklearn.decomposition.PCA
    2. Check the net
       1. If the weights have bad initialization
       2. If there is any overflow
          1. exp() is extremely easy to overflow:   
             sigmoid   
             when implementing GAN for semi-supervised learning,

## Where do we use MLP, CNN, RNN?

* + 1. MLP works well with data where structures don’t matter
    2. CNN works well with spatial data, such as images and videos
    3. RNN works well with sequential data, such as financial time series, text, translation, and audio signals

# Multi-Layer Perceptron (MLP)

## Neural network workflow

* + 1. Stochastic Gradient Descent  
       At each epoch, it first divides the inputs into batches, then it processes each batch of inputs as follows to refine the model:  
       1. It first processes the inputs through the network to produce outputs. Such a process is so called feedforward  
       2. After that the network use those output to compute the loss, which gives an insight into how far the current model is from the goal.   
       3. Now that it has the loss at hand, it’s time to improve the model: it calculates the partial derivatives with respect to each weight, and updates each weight by subtracting its corresponding partial derivatives multiplied by leaning rate. This process is known as backpropagation  
       At the end of each epoch, after all the batches are processed, the network runs through the validation set to see how the model is doing and if it’s overfitting

## Regression analysis

Regression analysis helps one understand how the typical value of the dependent variable (or criterion variable) changes when any one of the independent variables is varied, while the other independent variables are held fixed.

## Classification

* + 1. Perceptron

The perceptron is an algorithm for supervised learning(a task of inferring a function from labeled training data) of binary classifier (functions that can decide whether an input, represented by a vector of numbers, belongs to some specific class or not). It is a type of linear classifier, i.e. a classification algorithm that makes its predictions based on a linear predictor function combining a set of weights with the feature vector. The algorithm allows for online learning, in that it processes elements in the training set one at a time.



The step function, which simply maps the result of the linear function to 0/1, is used as the activation function

* + 1. What’s the connection between perceptron and gradient descent with sigmoid as activation function?
       1. The perceptron algorithm is in fact a special case of gradient descent, where ŷ can only be 0/1 as following picture shows

## Feedforward

The process neural networks use to turn the input into an output.

## Feedforward neural network (FNN)

A neural network wherein connections between the units do not form a cycle

## Deep neural network

A neural network whose hidden layer contains more than one layer

## How to improve the performance of MLP?

Unfamiliar point

* + 1. Simple model vs complicated model

Whenever we choose between a simple model that does the job and a complicated model that may do the job a little bit better, go for the simpler model

* + - 1. underfitting / error due to bias

The classifier is too simple, which fails to classify the data

* + - 1. overfitting / error due to variance

The classifier is too specific, which fails to generalize the model and thus do badly for data other than the training data

* + - 1. It’s hard to find the right architecture for a neural network, we always end up with either an overfitting one or an underfitting one. It’s better to go for the overfitting one and apply certain techniques to prevent overfitting on it
      2. When does overfitting likely happen?
         1. The dataset is small
      3. Techniques to prevent overfitting
         1. Early stopping

An algorithm to prevent overfitting. that is, we divide the data into two set: training set and validation set. Then we train the model using the training set and test the model using the validation set. The training ceases when the validation error reaches bottom

One downside of early stopping is that it kind of mixes optimizing loss function and preventing overfitting together. That violates an principle called orthogonalization which says that we want to be able to think about one task at a time. It makes the things we could try more complicated to think about

* + - * 1. Regularization

Any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error

* + - * 1. An concrete example:  
           Two models: x1+x2 and 10x1+10x2. The second one has better prediction for the training data (although they’re the same line in a diagram, the loss of the latter model is smaller than that of the first). But since 𝜎(10x1+10x2) is much steeper than 𝜎(x1+x2), it’s much harder to do gradient descent on it: the gradient are close to 0 for most cases and becomes very large when it gets to the middle of the curve
        2. The whole problem with Artificial Intelligence is that bad models are so certain of themselves, and good models so full of doubts

# Convolutional Neural Network (CNN)

## Convolution operation

Which applies a filter to an image

## Convolutional neural network (CNN)

A class of deep, feedforward neural network that has successfully been applied to analyzing visual imagery

* + 1. Class of hidden layers used in CNN
       1. Convolutional layer

A sparsely connected layer typically used to detect features

* + - * 1. filter / convolutional window

A grid containing weights. Each filter can be viewed as a pattern: the process calculating the value of a node in the convolutional layer from an input region is the process finding how much the input region is similar to the filter—the more the input region resembles the filter, the larger value the convolutional node has. As the training proceeds, filters are updated through backpropagation, and eventually able to represent some features.

As an example of filter

* + - * 1. Stride

The amount by which the filter slides over the image

* + - * 1. Padding

Which specify how to deal with the case that the filter extends outside the image

Valid

Get rid of these nodes

Same

Pad the image with 0 in advance to give the filter more space to move

* + - * 1. feature map

Which comprises of a set of nodes obtained after applying filter to the input. The reason calling it feature map is that the value of nodes indicates how much the input region resembles the filter, and a filter could be viewed as a pattern, or furthermore, a feature.

* + - * 1. For a volume of size (W, W, D), applying filter of size F, stride S, padding P, the size of the resultant feature map is  
           ((W - F + 2P)/S + 1, (W - F + 2P)/S + 1)
      1. Pooling layer

A layer typically used to decrease the spatial dimensions of feature map in the convolutional layer

recommended params: filter=2, stride=2

* + - * 1. Max pooling layer

It defines a window and stride as convolutional layer, and works as convolutional layer with two differences:

1. In channel calculations: The pooling window works with each feature map separately—so the resulting layer has the same channels as its previous layer, while convolutional windows work with all feature maps—so that the number of channels of the resulting layer hinges on the number of convolutional windows

2. In value calculations: Max pooling layers take the maximum value in the window as the value of the output node, where convolutional layers sum up the values in the window with respect to their weights

This is preferred in most cases, for that it extracts the most promising nodes (recall that the nodes with larger value in a feature map bears more resemblance to the filter)

An illustration of max pooling layer

* + - * 1. Average pooling layer

It’s almost like max pooling layer, except it takes the average value of nodes in the window instead of the maximum value

* + - * 1. Global pooling layer

It takes the average value of nodes in the whole feature map. It turns a three-dimensional array into a vector.

* + - * 1. For a volume of size (W, W, D), applying filter of size F, stride S, the size of the resultant feature map is  
           ((W - F) / S + 1, (W - F) / S + 1)
      1. After applying several convolutional and pooling layers, the output array, whose depth is much larger than its height and weight, contains no spatial information any more. Instead, it’s able to answer some feature questions, such as "is there a wheel", "are there eyes", and so on. That’s the time we can apply fully connected layers to make predictions

Convolutional layers extract features from the image

Pooling layers shrink feature maps to reduce the amount of computation and alleviate overfitting

Fully connected layers do high-level reasoning to make prediction

* + 1. What’s the advantage of CNN over MLP
       1. CNN accepts matrices as input so that it keeps spatial information when dealing with images
       2. CNN uses convolutional layers to reduce the number of parameters so that it alleviates vanishing gradient problem

## CNN layer patterns

* + 1. The most common CNN pattern
    2. Prefer a stack of small filter Conv layers to one with large filter
    3. To reduce memory requirement, make the compromise at the first layer — using filter of size 7 and stride of 2

## Invariant representation

* + 1. Scale invariance

The size of the object doesn’t affect the prediction

* + 1. Rotation invariance

The angle of the object doesn’t affect the prediction

* + 1. Translation invariance

The placement of the object doesn’t affect the prediction

* + 1. How to achieve the invariant representations?
       1. A technique called data augmentation helps achieve the invariant representations. It adds the corresponding resized/sheared/rotated/translated/so on images to the training set. This also helps with overfitting, because the model is trained by more data.

## What do filters in each layer generally detect?

* + 1. The filters in the first layer often detect edges or blobs of color  
       Those in the second layer might detects shapes, such as circles, stripes, rectangles, and so on  
       Those in the final layer detects more specific features to the data set, such as birds, dogs, wheels and so on

Layer mentioned in the above is not strictly means one layer, it may involves several layers.

All in all, the deeper filters are, the more specific features filters detect

## Assume we have a well-trained model at hand, and we are about to develop a new model, do we need to do this from scratch?

* + 1. In most cases, no. As discussed in [What do filters in each layer generally detect?](omnioutliner:///open?focus=k_sMMRA10bQ&row=kysLjO0QpED), there are several layers that might be common to both the old and new model. And that’s where transfer learning is introduced
    2. Transfer learning

Transfer learning involves taking a pre-trained neural network and adapting the neural network to a new, different data set.

* + - 1. Transfer learning is implemented depending on both:  
         1. The size of the new data set   
         2. The similarity of the new data to the original data set  
         Which results in four main cases:  
         1. New data set is small, new data is similar to original training data  
         2. New data set is small, new data is different from original training data   
         3. New data set is large, new data is similar to original training data  
         4. New data set is large, new data is different from original training data  
         The general strategy is illustrated as below
      2. Now consider an example. The pre-trained CNN is  
         1. For the case where the new data set is small and similar to the original training data:  
             1. slice off the end of the neural network  
             2. add a new fully connected layer that matches the number of classes in the new data set  
             3. randomize the weights of the new fully connected layer; freeze all the weights from the pre-trained network  
             4. train the network to update the weights of the new fully connected layer

To avoid overfitting on the small data set, the weights of the original network will be held constant rather than re-training the weights.

Since the data sets are similar, images from each data set will have similar higher level features. Therefore most or all of the pre-trained neural network layers already contain relevant information about the new data set and should be kept.

* + - * 1. For the case where the new data is small and different from the original training data  
            1. slice off most of the pre-trained layers near the beginning of the network  
            2. add to the remaining pre-trained layers a new fully connected layer that matches the number of classes in the new data set  
            3. randomize the weights of the new fully connected layer; freeze all the weights from the pre-trained network  
            4. train the network to update the weights of the new fully connected layer

Because the data set is small, overfitting is still a concern. To combat overfitting, the weights of the original neural network will be held constant, like in the first case.

But the original training set and the new data set do not share higher level features. In this case, the new network will only use the layers containing lower level features.

* + - * 1. For the case where the new data set is large and similar to the original training data:  
            1. remove the last fully connected layer and replace with a layer matching the number of classes in the new data set  
            2. randomly initialize the weights in the new fully connected layer  
            3. initialize the rest of the weights using the pre-trained weights  
            4. re-train the entire neural network

Overfitting is not as much of a concern when training on a large data set; therefore, you can re-train all of the weights.

Because the original training set and the new data set share higher level features, the entire neural network is used as well.

* + - * 1. If the new data set is large and different from the original training data:  
            1. remove the last fully connected layer and replace with a layer matching the number of classes in the new data set  
            2. retrain the network from scratch with randomly initialized weights  
            3. alternatively, you could just use the same strategy as the "large and similar" data case

## How to load an image from a file path

* + 1. from keras.preprocessing import image  
       def read\_img(path, target\_size = None):  
        # load the image into PIL format  
        img = image.load\_img(path, target\_size)  
        # convert a PIL instance to a Numpy array, a 3D tensor  
        return image.img\_to\_array(img)

path str: path to image file

target\_size [int, int]: tuple of ints `height, width`. Default is`None`, which is the original image size

* + 1. import cv2  
       def read\_img(path, target\_size = None):  
        # load the image as a Numpy array, a 3D tensor  
        img = cv2.imread(path)  
        if target\_size == None:  
        return img  
        return cv2.resize(img, target\_size)

## How to plot an image?

* + 1. import matplotlib.pyplot as plt  
       import cv2  
       img = cv2.imread('1.jpg')  
       # for the original image read by cv2 is in BGR model, but plt follows RGB order. Convert it in RGB model   
       img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)  
       plt.imshow(img)  
       plt.show()

Another way to convert an BGR image into RGB model is:

img[…, ::-1]

For image is a numpy.ndarray of form (height, width, channel) and the channel dimension represents color order

## CNN architectures

* + 1. AlexNet first use CNN
    2. VGG uses smaller filter size and deeper model
    3. GoogLeNet uses inception module

GoogLeNet was published before the invention of batch normalization, it resorted to auxiliary classifiers to add extra gradients so as to avoid vanish gradient problems

* + - 1. 1. GoogLeNet is computationally expansive. Inception module employs bottleneck layer (1 x 1 conv) to reduce depth so as to mitigate computational intensity.
    1. ResNet uses residual connections and very deep model
       1. Motivated by deeper networks are hards to train but we wish the deeper model performs at least as well as the shallow model
       2. Residual blocks can be ignored by learning all the parameters zero.
       3. Residual connections helps gradient flow backward by providing additional direct highway
    2. Highway Network modifies ResNet in the way that it resorts to an extra conv layer with sigmoid as the activation function (T(X)) to weigh X and F(X) — T(X)F(X)+(1-T(X))X
    3. DenseNet concatenate instead of adding F(X) and X

## Further readings

* + 1. If you would like to know more about interpreting CNNs and convolutional layers in particular, you are encouraged to check out these resources:  
        • Here's a [section](http://cs231n.github.io/understanding-cnn/) from the Stanford's CS231n course on visualizing what CNNs learn.  
        • Check out this [demonstration](https://aiexperiments.withgoogle.com/what-neural-nets-see) of a cool [OpenFrameworks](http://openframeworks.cc/) app that visualizes CNNs in real-time, from user-supplied video!  
        • Here's a [demonstration](https://www.youtube.com/watch?v=AgkfIQ4IGaM&t=78s) of another visualization tool for CNNs. If you'd like to learn more about how these visualizations are made, check out this [video](https://www.youtube.com/watch?v=ghEmQSxT6tw&t=5s).  
        • Here's [another visualization tool](https://medium.com/merantix/picasso-a-free-open-source-visualizer-for-cnns-d8ed3a35cfc5) that seamlessly works with CNNs in Keras and Tensorflow.  
        • Read this [Keras blog post](https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html) on visualizing how CNNs see the world. In this post, you can find an accessible introduction to Deep Dreams, along with code for writing your own deep dreams in Keras. When you've read that:  
        • Also check out this [music video](https://www.youtube.com/watch?v=XatXy6ZhKZw) that makes use of Deep Dreams (look at 3:15-3:40)!  
        • Create your own Deep Dreams (without writing any code!) using this [website](https://deepdreamgenerator.com/).  
        • If you'd like to read more about interpretability of CNNs,  
        • here's an [article](https://blog.openai.com/adversarial-example-research/) that details some dangers from using deep learning models (that are not yet interpretable) in real-world applications.  
        • there's a lot of active research in this area. [These authors](https://arxiv.org/abs/1611.03530) recently made a step in the right direction.
    2. • <https://www.kaggle.com/c/imagenet-object-detection-challenge>  
        • <https://www.kaggle.com/c/imagenet-object-detection-from-video-challenge>  
        • <https://www.kaggle.com/c/nips-2017-defense-against-adversarial-attack>  
        • <https://medium.com/towards-data-science/transfer-learning-using-keras-d804b2e04ef8>  
        • <http://cs231n.github.io/transfer-learning/>  
        • <http://www.slate.com/articles/technology/future_tense/2016/04/the_philosophical_argument_against_artificial_intelligence_killing_us_all.html>  
        • <http://www.robots.ox.ac.uk/~vgg/publications/2015/Parkhi15/parkhi15.pdf>

## Visualizing

* + 1. Weights in the first layer can be visualized to show some information of color, bubbles, shape, etc.
    2. Features in the last layer give us a way to see what’s going
       1. Applying nearest neighbors on these feature vectors can show us what kind of images is analogous to the test image
       2. T-SNE provides another way to classify images based on the features in the last layer
    3. Maximally activating patches: The receptive field for a neuron in a activation map
    4. Occlusion experiments: mask part of the image before classification to see how it influences the classification result
    5. Saliency maps: compute the gradients of the class score w.r.t. image pixels, take absolution value and max over RGB channels. This tells us which pixels in the image matters for the classification — in the first order sense, for each pixel in the image, if we wiggle the pixel a little bit, how much the classification will change
    6. Gradient accent: apply gradient accent to maximize a neuron

# Recurrent Neural Network(RNN)

## Folded view & unfolded view of recursion

* + 1. The unfolded view specifies individually how each element in a sequence is generated
    2. The folded view gives a summary pattern that describes how each element in a sequence is generated

## Driver & hidden sequence

* + 1. Driver is the sequence that drives another sequence

Driver is received as data

* + 1. Hidden sequence is the sequence driven by driver

Hidden sequence is generated recursively using the driver

* + 1. An example  
       h{0} = 0  
       h{i} = h{i-1} + s{i-1}  
       Here, sequence s drives sequence h, so s is the driver, whereas h is the hidden sequence

## Order

The order of a sequence is the number of previous elements needed to generate a new value

## How to use FNN to model sequential data?

* + 1. We expect that each element in the sequence (s{1}, …) can be calculated from its previous elements. Here, without loss of generality, we just assume the order is one, i.e., each element is calculated from the one before it. It has form:  
       h{1} = a  
       h{t} = g(s{t-1})  
       (Note what FNN does is to make h approximate s as well as possible) The error for each level t is   
       (s{t} - h{t})^2.   
       Then we sum them up to get a total loss function:   
       ∑(s{t} - h{t})^2  
       The FNN tunes the parameters of g to minimize this least squares loss

## Independently and identically distributed / IDD

A series of variables are IDD iff each random variable has the same probability distribution as the others and all are mutually independent

* + 1. Heads up: n values generated by a series of n IID variables could be viewed as sampling a single variable n times
    2. When we use values generated by a series of IDD variables to make an input/output plot, in which x are the values of the variables in the sequence and each x’s corresponding y is the value of the variable following that x, the input/output plot has following attributes:  
       I. The number of cluster in the input/output plot is the square of the number of humps in the probability distribution.  
       II. If the probability distribution is symmetric, the input/output plot is symmetric along the x-axis and the y-axis (shifted according to the position of the axis of symmetry in the probability distribution)  
       III. The clusters is equally represented to the extent that they’re represented by the probability distribution

If we regard the I/O plot as sampling x twice, things become clear

* + - 1. For example, consider the probability distribution is as below  
          The corresponding input/output plot should have 9 clusters, here I list them according to the corresponding population:  
          The most populated is the one centered at (x3, x3)  
          The second are at (x2, x3) and (x3, x2)  
          Then (x1, x3), (x3, x1), and (x2, x2)  
          And (x1, x2) and (x2, x1)  
          The least populated is centered at (x1, x2)

## Why RNN? Why not just use FNN to model sequential data?

* + 1. There are some fatal flaws about using FNN to model sequential data:   
       I. The FNN model doesn’t introduce any dependence between different levels, that is, h{t} is independent of all previous h{i} for i < t — it only depends on s{t-1}, which is an observed variable.  
       II. When we use FNN, we’re trying to approximate the data with a recursive sequence. That is, we wish there is some dependency between the data. Least squares, however, assumes (input, output) pairs are IID, in this case, (s{t-1}, s{t})s are IID. This violates our initial attempt
    2. RNN introduces dependence between consecutive levels. It calculates each element based on both the values of elements before it and the results of previous calculations. For the case where the order is one, it has form:  
       h{1} = s{1}  
       h{t} = f(h{t-1}, s{t-1})  
       We call these h{}s hidden states. Through this definition, it’s easy to see that every hidden state contains a complete history of the sequence of elements that precede it, which is so-called memory.  
       Moreover, now that the (input, output) pairs becomes ((h{t-1}, s{t-1}), s{t})s, which is truly IID, there is no conflict to use least squares

Here the sequence s is the driver, the sequence h is the hidden sequence

## Image Caption Workflow



## LSTM

Long short term memory, a RNN architecture.

* + 1. We’ve already got RNNs to have memory of previous information, why do we need LSTM?
       1. In RNNs, the old memory fades as new memory involves — the vanishing gradient occurs because of the multiplication happening between levels. The memory it stores is chaotic — it doesn’t distinguish the important from the trivial. That’s why we introduce LSTM to distill the nontrivial so as to store long term memory as well short one
       2. The long term memory in LSTM doesn’t suffer the vanishing gradient problem, because the operations between levels are addition instead of multiplication
    2. Architecture

       2. Inputs
          1. Long term memory
          2. Short term memory
          3. Events
       3. Summary

This is an arbitrary architecture, why it is this case simply because it works well. Others may also do the job.

LTM is also called cell state, a vector

STM hidden state, a vector

E is the input, a vector

𝜎s and tanhs are neural network layers, with sigmoid and tanh as activation functions, respectively

𝜎s suggest how much of a term should be retained. From left to right, they are named:

Forget gate (despite the fact that a 1 in the forget gate still means to keep the memory and a 0 still means to forget it)

Input gate (as it determines how much of the input to let into the cell state)

Output gate (as it decides which part of cell state we’re going to output, and save as hidden state)

tanhs squash the value to be between -1 and 1

Multiplications and additions here are pointwise operations. From left to right, what these multiplications do are:

Extracting the useful information in the cell state, dropping the useless

Extracting the useful information in the input and hidden state, dropping the useless

Extracting the information from the updated cell state that will be immediately useful to form a new hidden state

What the addition does is:

Combining the useful information in the input, hidden state and old cell state to construct a new cell state

* + - * 1. Step-by-step walk through

The forget gate decides what should be retained in the cell state

The first tanh creates a vector of new candidate values, and the input gate decides what to let into the cell state

The cell state drops the things decided to forget earlier (×), and then adds those candidate values decided to let into (+)

The second tanh then push the values in cell state between -1 and 1, and the output gate decides what to output

* + - 1. Variants

It adds the peephole connection to gates, to let them look at the cell state



Instead of separately deciding what to forget and what to add, it makes those decisions together: add those we’re going to forget, and forget those we’re going to add

* + 1. Further readings
       1. [This article](http://colah.github.io/posts/2015-08-Understanding-LSTMs/) elucidates LSTM in detail. Moreover, it enumerates some of its variants that share the same idea in different form. Last but not least, it provides some other interesting information, in conclusion, to study RNN in depth, such as [attention](https://arxiv.org/pdf/1502.03044v2.pdf), [Grid LSTM](https://arxiv.org/pdf/1507.01526v1.pdf), etc.

## Project Anna\_KaRNNa: A text generator that generates new text based on the training text

* + 1. Preprocessing the training data
       1. Divide the training text into several batches. Each batch contains several(minibatch size) sequences, which run simultaneously to speed up the computation

* + - * 1. def get\_batches(arr, n\_seqs, n\_steps):  
            '''  
            Paras:  
            arr []: input sequential data  
            n\_seqs: number of sequences per batch  
            n\_steps: number of steps per sequence  
            Return:  
            [features, targets]: devided batches of features and targets  
            '''  
            # trim the data to make sure batches are full  
            characters\_per\_batch = n\_seqs \* n\_steps  
            n\_batches = len(arr) // characters\_per\_batch  
            arr = arr[: n\_batches \* characters\_per\_batch]  
            # make sure sequences between consecutive batches are connected in the original context  
            arr = arr.reshape((n\_seqs, -1)  
               # yield batches of feature, targets  
            for i in range(0, arr.shape[1], n\_steps):  
            features = arr[:, i: i + n\_steps]  
            targets = np.concatenate((features[:, 1:], features[:, 0]), axis=1)  
            yield features, targets

Notice how this process divides the array—it divides the array into n\_seqs instead of n\_batches. In this way , it establishes connection between continuous batches so as to maintain continuity of each sequence even when sub-sequences in batch get trained simultaneously

* + 1. Building the model
       1. Process Overview

* + - * 1. class CharRNN:  
              
            def \_\_init\_\_(self, num\_classes, batch\_size=64, num\_steps=50,   
            lstm\_size=128, num\_layers=2, learning\_rate=0.001,   
            grad\_clip=5):  
              
            # When we're using this network for sampling later, we'll be passing in  
            # one character at a time, so providing an option for that  
            if sampling == True:  
            batch\_size, num\_steps = 1, 1  
            else:  
            batch\_size, num\_steps = batch\_size, num\_steps  
            tf.reset\_default\_graph()  
              
            # Build the input placeholder tensors  
            self.inputs, self.targets, self.keep\_prob = build\_inputs(batch\_size, num\_steps)  
            # Build the LSTM cell  
            cell, self.initial\_state = build\_lstm(lstm\_size, num\_layers, batch\_size, self.keep\_prob)  
            ### Run the data through the RNN layers  
            # First, one-hot encode the input tokens  
            x\_one\_hot = tf.one\_hot(self.inputs, num\_classes)  
              
            # Run each sequence step through the RNN and collect the outputs  
            outputs, state = tf.nn.dynamic\_rnn(cell, x\_one\_hot, initial\_state=self.initial\_state)  
            self.final\_state = state  
              
            # Get softmax predictions and logits  
            self.prediction, self.logits = build\_output(outputs, lstm\_size, num\_classes)  
              
            # Loss and optimizer (with gradient clipping)  
            self.loss = build\_loss(self.logits, self.targets, num\_classes)  
            self.optimizer = build\_optimizer(self.loss, learning\_rate, grad\_clip)

To run data through hidden layers, we call tf.nn.dynamic\_rnn instead of calling cell itself because tf.nn.dynamic\_rnn is more efficient in that it uses a loop to dynamically construct the graph:

tf.nn.rnn creates an unrolled graph for a fixed RNN length. That means, if you call tf.nn.rnn with inputs having 200 time steps you are creating a static graph with 200 RNN steps. First, graph creation is slow. Second, you’re unable to pass in longer sequences (> 200) than you’ve originally specified.

tf.nn.dynamic\_rnn solves this. It uses a tf.wthile\_loop to dynamically construct the graph when it is executed. That means graph creation is faster and you can feed batches of variable size.

* + - 1. Create placeholders for the inputs
         1. def build\_inputs(self, batch\_size, num\_steps):  
             """ Build the input placeholders  
             :param batch\_size: Number of sequences per batch  
             :param num\_steps: Number of steps per batch  
             :return: tf.placeholder for features, labels, and keep probability  
             """  
             features = tf.placeholder(tf.int32, (batch\_size, num\_steps), name='features')  
             labels = tf.placeholder(tf.int32, (batch\_size, num\_steps), name='labels')  
             keep\_prob = tf.placeholder(tf.float32, name='keep\_prob')  
             return features, labels, keep\_prob

Input and target shape: [batch\_size, num\_steps] — batch\_size specifies the number of sequences which could run simultaneously during the process, num\_steps specifies the length of the sequence

* + - 1. Build hidden layers (LSTM cell)
         1. def build\_lstm(self, num\_units, num\_layers, batch\_size, keep\_prob):  
             """ Build LSTM layer  
             :param num\_units: Number of hidden units  
             :param num\_layer: Number of LSTM layers  
             :param batch\_size: Number of sequences per batch  
             :param keep\_prob: Keep probability  
             :return: LSTM cell and initial state  
             """  
             def cell(num\_units, keep\_prob):  
             cell = tf.contrib.rnn.BasicLSTMCell(num\_units)  
             dropout = tf.contrib.rnn.DropoutWrapper(cell, output\_keep\_prob=keep\_prob)  
               
             return dropout  
             cells = tf.contrib.rnn.MultiRNNCell([cell(num\_units, keep\_prob) for \_ in num\_layers])  
             initial\_state = cells.zero\_state(batch\_size, tf.float32)  
               
             return cells, initial\_state

To construct a RNN cell, we have to specify the number of hidden units

For multiple RNN cell, we have to specify the number of layers

Last but not least, we have to specify batch size to construct the initial state

The initial state is the initial long term memory, which is initialized with zeros and gets updated during training

* + - 1. Build output layer
         1. def build\_output(self, lstm\_output, input\_size, output\_size):  
             """ Build output softmax layer  
             :param lstm\_output: Tensor, output of hidden layer  
             :param input\_size: Size of the input tensor, i.e. the number of hidden units  
             :param output\_size: Size of this softmax layer  
             :return: probs, logits: Probabilities and logits  
             """  
             # Reshape output so it's a bunch of rows, one row for each step for each sequence.  
             # That is, the shape should be batch\_size\*num\_steps rows times lstm\_size columns  
             x = tf.reshape(lstm\_output, (-1, input\_size))  
             logit = tf.contrib.layers.fully\_connected(x, output\_size, activation\_fn=None)  
             predictions = tf.nn.softmax(logits, name='predictions')  
             return predictions, logits
         2. What’s the output of the LSTM cells?

A tensor with size batch\_size\*num\_steps\*num\_units. The output of each LSTM cell has num\_units hidden units, we have num\_steps of them, one for each sequence step, and we have batch\_size sequences.

* + - 1. Build loss
         1. def build\_loss(self, logits, labels, num\_classes):  
             """ Build loss function  
             :param logits: Logits  
             :param labels: Labels for supervised learning  
             :param num\_classes: Number of classes in targets  
             :return:  
             """  
             y\_one\_hot = tf.one\_hot(labels, num\_classes)  
             y\_one\_hot = tf.reshape(y\_one\_hot, logits.get\_shape())  
             loss = tf.nn.softmax\_cross\_entropy\_with\_logits(labels=y\_one\_hot, logits=logits)  
             loss = tf.reduce\_mean(loss)  
             return loss
      2. Build optimizer
         1. def build\_optimizer(self, loss, learning\_rate, grad\_clid):  
             """ Build optmizer for training, using gradient clipping.  
             :param grad\_clid:  
             :return:  
             """  
             # Optimizer for training, using gradient clipping to control exploding gradients  
             tvars = tf.trainable\_variables()  
             grads, \_ = tf.clip\_by\_global\_norm(tf.gradients(loss, tvars), grad\_clip)  
             train\_op = tf.train.AdamOptimizer(learning\_rate)  
             optimizer = train\_op.apply\_gradients(zip(grads, tvars))  
             return optimizer

## Further readings

* + 1. [this great paper on gradient clipping](http://proceedings.mlr.press/v28/pascanu13.pdf) to avoid vanishing / exploding gradient
    2. As it appears, you have some idea of LSTMs & RNNs, here’s a very [popular blog](http://colah.github.io/posts/2015-08-Understanding-LSTMs/) that might help you in visually understanding further details.  
       Advanced tips for improving net results  
        • Try and use deeper architectures, which have general tendency to blow up or vanish the gradients - so there's a net architecture known as Residual Nets, used to circumnavigate the issues with deeper architectures  
        • Try using more fully connected layers or Bi-Directional LSTMs or GRUs to make the predictions even better  
        • Try and use more sophisticated methods like lemmatisation and stemming to create a more pruned vocabulary. Have a look at the [NLTK](http://www.nltk.org/) library to understand more operations  
       If you are keen on learning a bit more into what Natural Language Scientists use regularly in their nets. Try reading up a bit more on  
        • Word2Vec Algorithm  
        • Glove Algorithm  
        • [Sequence2Sequence tutorial](https://www.tensorflow.org/tutorials/seq2seq)  
       Keep up the good work !

## LSTM vs GRU

* + 1. There is no clear judgement about which cell is better. Try both on a random subset of the training set to find out.
    2. 2 is the recommended number of layers for RNN

# Generative Adversarial Network (GAN)

Generative adversarial network

## Autoencoder

A neural network used for unsupervised learning (the task of inferring a function to describe hidden structure from "unlabeled" data (a classification or categorization is not included in the observations)) of efficient codings

* + 1. The structure of autoencoder
    2. When is autoencoder useful?
       1. Autoencoder is useful to denoise images. It extracts features from the noisy image (encoding), and reconstruct images from these features (decoding). During encoding process, noise wears off gradually, and such noise won’t be reconstructed again in decoding process
       2. Autoencoder can also be used to compression, but it does worse than traditional methods
       3. Autoencoder can be used to extract low-dimensional features
       4. Encoder can be used to initialize a supervised model
    3. Encoder
       1. We use convolutional layer to extract features and max pooling layer to do downsampling
       2. At the end we get a layer of which depth is large, height and weight are small. That’s the code in above structure figure
    4. Decoder
       1. We use convolutional layer as we do in encoder and resize the layer (using [nearest-neighbor interpolation](https://en.wikipedia.org/wiki/Nearest-neighbor_interpolation) or [bilinear interpolation](https://en.wikipedia.org/wiki/Bilinear_interpolation)) to do upsampling
       2. Convolutional layer and resizing layer are arranged as convolutional layer and max pooling layer were when encoding, but in reverse order.
    5. An example with tensorflow
       1. ### Encoder  
          # Original 28x28x1  
          conv1 = tf.layers.conv2d(inputs\_, 16, (3,3), padding='same', activation=tf.nn.relu)  
          # Now 28x28x16  
          maxpool1 = tf.layers.max\_pooling2d(conv1, (2,2), (2,2), padding='same')  
          # Now 14x14x16  
          conv2 = tf.layers.conv2d(maxpool1, 8, (3,3), padding='same', activation=tf.nn.relu)  
          # Now 14x14x8  
          maxpool2 = tf.layers.max\_pooling2d(conv2, (2,2), (2,2), padding='same')  
          # Now 7x7x8  
          conv3 = tf.layers.conv2d(maxpool2, 8, (3,3), padding='same', activation=tf.nn.relu)  
          # Now 7x7x8  
          encoded = tf.layers.max\_pooling2d(conv3, (2,2), (2,2), padding='same')  
          # Now 4x4x8  
          ### Decoder  
          upsample1 = tf.image.resize\_nearest\_neighbor(encoded, (7,7))  
          # Now 7x7x8  
          conv4 = tf.layers.conv2d(upsample1, 8, (3,3), padding='same', activation=tf.nn.relu)  
          # Now 7x7x8  
          upsample2 = tf.image.resize\_nearest\_neighbor(conv4, (14,14))  
          # Now 14x14x8  
          conv5 = tf.layers.conv2d(upsample2, 8, (3,3), padding='same', activation=tf.nn.relu)  
          # Now 14x14x8  
          upsample3 = tf.image.resize\_nearest\_neighbor(conv5, (28,28))  
          # Now 28x28x8  
          conv6 = tf.layers.conv2d(upsample3, 16, (3,3), padding='same', activation=tf.nn.relu)  
          # Now 28x28x16  
          logits = tf.layers.conv2d(conv6, 1, (3,3), padding='same', activation=None)  
          #Now 28x28x1
    6. Denoising autoencoder

In which we add noise to the input and expect the autoencoder to reconstruct the image

* + 1. Contractive autoencoder

In which we add the squared Frobenius norm of the Jacobian matrix J(x) to the loss so as to penalize local perturbation of the inputs

* + - 1. Compare to denoising autoencoders: denoising autoencoders make the reconstruction function resist small but finite-sized perturbations of the input, while contractive autoencoders make the feature extraction function resist infinitesimal perturbations of the input
    1. Adversarial autoencoder
       1. Add labels to the input of the discriminator and use a mixture of Gaussians, one for each class
    2. The latent code is only comprised of the most prominent variation in X and leave the decoder to fill in the common details

## What can GAN do?

* + 1. GAN, based on previous learning, generates things, which is real enough to fool a discriminator. Because of this property, it can be used to generate photos, videos. Moreover, it can even simulate physical experiments

## Generative model

* + 1. Explicit density estimation: explicitly define and solve the input distribution P(X)
    2. Implicit density estimation: learn model that can sample from P(X)
    3. Generative models of time-series data can be used for simulation and planning in reinforcement learning application
    4. Pixel RNN is slow on training and generating time
    5. Pixel CNN is fast on training time but slow on generating time

## What does GAN consist of?

* + 1. It consists of a generator and a discriminator

Discriminator distinguishes real images from generated images. It’s trained by real&generated images

Generator generates images, and tries to use those images to fool the discriminator. It gets trained through discriminator by setting variables in discriminator fixed

* + 1. Models for generator and discriminator

The discriminators in generator model and discriminator model share the same variables

* + - 1. Generator:
      2. Discriminator:
    1. We’ll have a favor to one over another for different purposes. For example, if we use GAN to generate images, we usually throw away the discriminator at the end of the training, and use generator to generate images. If we use GAN for semi-supervised learning, we usually throw away the generator after the training is done, and use discriminator as a classifier.

## Activation function used in GAN

* + 1. For hidden layers: lrelu, i.e. leaky relu, which is like normal relu, except that there is a small non-zero output for negative input values

lrelu(x) = max(a\*x, x), where a is a small decimal

In tensorflow, it can be written as

layer = tf.maximum(layer \* alpha, layer)

* + 1. For output layer of generator: tanh
    2. For output layer of discriminator: sigmoid

## Loss functions in GAN

* + 1. Discriminator tries to distinguish true data from fake data, while generator tries to maxi
    2. Label smoothing

When we calculate the discriminator loss for the real image, we don’t set all the labels to be one, instead, we set them to be somewhat less than 1, 0.9 for example.

* + 1. How does label smoothing help the discriminator generalize better?
       1. It prevents the discriminator overfitting, and thus helps the discriminator to generalize better and avoid learning to make extreme predictions when extrapolating

## GAN for semi-supervised learning

* + 1. Output Classes
       1. The output classes are comprised of two part:  
          The first part consists of all the categories of real images, as defined by general classifiers  
          The second part is a class for fake images, indicates the probability that the image is fake
    2. Loss function for discriminator
       1. Total loss = loss for labeled + loss for unlabeled
       2. Loss for labeled = softmax cross-entropy(logits, class labels)
       3. Loss for unlabeld = sigmoid cross-entropy(logits, real labels)

For unlabeled data (which include unlabeled real data and generated data), we calculate the real probability by summing up the probabilities of the first part of the output, and use this probability along with the probability of the second part to compute the loss via cross-entropy

* + - * 1. How to calculate the logits for unlabeled data that indicate data are real?

sigmoid(logits\_to\_cal) = 1 - softmax(fake\_class\_logits)

logits\_to\_cal and x are the logits we’re about to calculate

fake\_class\_logits and f are the logits of the fake class which can be directly obtained from the network

i in the sum stands for the logits of each of the well categorized real image

* + 1. Loss function for generator
       1. We add an extra term to the loss function for the generator, penalizing the mean absolute error between the average value of some set of features on the training data and the average value of that set of features on the generated samples. The set of features can be any group of hidden units from the discriminator
          1. x7 = tf.layers.conv2d(relu5, 2 \* size\_mult, 3, strides=1, padding='valid')  
             ''' This is the layer we want to use to calculate the average value of features  
             so don't use batch normalization here  
             '''  
             relu7 = tf.maximum(alpha \* x7, x7)  
                
             ''' Extract the features:  
             Flatten it by global average pooling  
             (1, 2) is the spatial domain (height and width) of the feature map  
             '''  
             features = tf.reduce\_mean(relu7, (1, 2))
    2. An example for discriminator
       1. def discriminator(x, reuse=False, alpha=0.2, drop\_rate=0., num\_classes=10, size\_mult=64):  
           with tf.variable\_scope('discriminator', reuse=reuse):  
           x = tf.layers.dropout(x, rate=drop\_rate/2.5)  
             
           # Input layer is 32x32x3  
           x1 = tf.layers.conv2d(x, size\_mult, 3, strides=2, padding='same')  
           relu1 = tf.maximum(alpha \* x1, x1)  
           relu1 = tf.layers.dropout(relu1, rate=drop\_rate)  
             
           x2 = tf.layers.conv2d(relu1, size\_mult, 3, strides=2, padding='same')  
           bn2 = tf.layers.batch\_normalization(x2, training=True)  
           relu2 = tf.maximum(alpha \* x2, x2)  
             
             
           x3 = tf.layers.conv2d(relu2, size\_mult, 3, strides=2, padding='same')  
           bn3 = tf.layers.batch\_normalization(x3, training=True)  
           relu3 = tf.maximum(alpha \* bn3, bn3)  
           relu3 = tf.layers.dropout(relu3, rate=drop\_rate)  
             
           x4 = tf.layers.conv2d(relu3, 2 \* size\_mult, 3, strides=1, padding='same')  
           bn4 = tf.layers.batch\_normalization(x4, training=True)  
           relu4 = tf.maximum(alpha \* bn4, bn4)  
             
           x5 = tf.layers.conv2d(relu4, 2 \* size\_mult, 3, strides=1, padding='same')  
           bn5 = tf.layers.batch\_normalization(x5, training=True)  
           relu5 = tf.maximum(alpha \* bn5, bn5)  
             
           x6 = tf.layers.conv2d(relu5, 2 \* size\_mult, 3, strides=2, padding='same')  
           bn6 = tf.layers.batch\_normalization(x6, training=True)  
           relu6 = tf.maximum(alpha \* bn6, bn6)  
           relu6 = tf.layers.dropout(relu6, rate=drop\_rate)  
             
           ''' followings are the noteworthy part '''  
           x7 = tf.layers.conv2d(relu5, 2 \* size\_mult, 3, strides=1, padding='valid')  
           # This is the layer we want to use to calculate the average value of features  
           # So don't apply batch normalization here  
           relu7 = tf.maximum(alpha \* x7, x7)  
             
           # Extract the features:  
           # Flatten it by global average pooling  
           # (1, 2) is the height and width of relu7  
           features = tf.reduce\_mean(relu7, (1, 2))  
             
           # Set class\_logits to be the inputs to a softmax distribution over the different classes  
           class\_logits = tf.layers.dense(features, num\_classes + extra\_class)  
             
             
           # Set gan\_logits such that P(input is real | input) = sigmoid(gan\_logits).  
           # Keep in mind that class\_logits gives you the probability distribution over all the real  
           # classes and the fake class. You need to work out how to transform this multiclass softmax  
           # distribution into a binary real-vs-fake decision that can be described with a sigmoid.  
           # Numerical stability is very important.  
           # You'll probably need to use this numerical stability trick:  
           # log sum\_i exp a\_i = m + log sum\_i exp(a\_i - m).  
           # This is numerically stable when m = max\_i a\_i.  
           # (It helps to think about what goes wrong when...  
           # 1. One value of a\_i is very large, then exp(a\_i) overflows  
           # 2. All the values of a\_i are very negative, then sum\_i underflows  
           # This trick and this value of m fix both those cases, but the naive implementation and  
           # other values of m encounter various problems)  
             
           if extra\_class:  
           real\_class\_logits, fake\_class\_logits = tf.split(class\_logits, [num\_classes, 1], 1)  
           assert fake\_class\_logits.get\_shape()[1] == 1, fake\_class\_logits.get\_shape()  
           fake\_class\_logits = tf.squeeze(fake\_class\_logits)  
           else:  
           real\_class\_logits = class\_logits  
           fake\_class\_logits = 0.  
             
           mx = tf.reduce\_max(real\_class\_logits, 1, keep\_dims=True)  
           stable\_real\_class\_logits = real\_class\_logits - mx  
           gan\_logits = tf.log(tf.reduce\_sum(tf.exp(stable\_real\_class\_logits), 1)) + tf.squeeze(mx) - fake\_class\_logits  
             
           out = tf.nn.softmax(class\_logits)  
             
           return out, class\_logits, gan\_logits, features
    3. An example for loss functions
       1. def model\_loss(input\_real, input\_z, output\_dim, y, num\_classes, label\_mask, alpha=0.2, drop\_rate=0.):  
           """  
           Get the loss for the discriminator and generator  
           :param input\_real: Images from the real dataset  
           :param input\_z: Z input  
           :param output\_dim: The number of channels in the output image  
           :param y: Integer class labels  
           :param num\_classes: The number of classes  
           :param alpha: The slope of the left half of leaky ReLU activation  
           :param drop\_rate: The probability of dropping a hidden unit  
           :return: A tuple of (discriminator loss, generator loss)  
           """  
             
             
           # These numbers multiply the size of each layer of the generator and the discriminator,  
           # respectively. You can reduce them to run your code faster for debugging purposes.  
           g\_size\_mult = 32  
           d\_size\_mult = 64  
             
           # Here we run the generator and the discriminator  
           g\_model = generator(input\_z, output\_dim, alpha=alpha, size\_mult=g\_size\_mult)  
           d\_on\_data = discriminator(input\_real, alpha=alpha, drop\_rate=drop\_rate, size\_mult=d\_size\_mult)  
           d\_model\_real, class\_logits\_on\_data, gan\_logits\_on\_data, data\_features = d\_on\_data  
           d\_on\_samples = discriminator(g\_model, reuse=True, alpha=alpha, drop\_rate=drop\_rate, size\_mult=d\_size\_mult)  
           d\_model\_fake, class\_logits\_on\_samples, gan\_logits\_on\_samples, sample\_features = d\_on\_samples  
             
             
           # Here we compute `d\_loss`, the loss for the discriminator.  
           # This should combine two different losses:  
           # 1. The loss for the GAN problem, where we minimize the cross-entropy for the binary  
           # real-vs-fake classification problem.  
           # 2. The loss for the SVHN digit classification problem, where we minimize the cross-entropy  
           # for the multi-class softmax. For this one we use the labels. Don't forget to ignore  
           # use `label\_mask` to ignore the examples that we are pretending are unlabeled for the  
           # semi-supervised learning problem.  
           d\_loss\_real = tf.reduce\_mean(  
           tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits=gan\_logits\_on\_data,  
           labels=tf.ones\_like(gan\_logits\_on\_data)))  
           d\_loss\_fake = tf.reduce\_mean(  
           tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits=gan\_logits\_on\_samples,  
           labels=tf.zeros\_like(gan\_logits\_on\_samples)))  
           y = tf.squeeze(y)  
           class\_cross\_entropy = tf.nn.softmax\_cross\_entropy\_with\_logits(logits=class\_logits\_on\_data,  
           labels=tf.one\_hot(y, num\_classes + extra\_class,  
           dtype=tf.float32))  
           class\_cross\_entropy = tf.squeeze(class\_cross\_entropy)  
           label\_mask = tf.squeeze(tf.to\_float(label\_mask))  
           d\_loss\_class = tf.reduce\_sum(label\_mask \* class\_cross\_entropy) / tf.maximum(1., tf.reduce\_sum(label\_mask))  
           d\_loss = d\_loss\_class + d\_loss\_real + d\_loss\_fake  
             
           # Here we set `g\_loss` to the "feature matching" loss invented by Tim Salimans at OpenAI.  
           # This loss consists of minimizing the absolute difference between the expected features  
           # on the data and the expected features on the generated samples.  
           # This loss works better for semi-supervised learning than the tradition GAN losses.  
           data\_moments = tf.reduce\_mean(data\_features, axis=0)  
           sample\_moments = tf.reduce\_mean(sample\_features, axis=0)  
           g\_loss = tf.reduce\_mean(tf.abs(data\_moments - sample\_moments))  
           pred\_class = tf.cast(tf.argmax(class\_logits\_on\_data, 1), tf.int32)  
           eq = tf.equal(tf.squeeze(y), pred\_class)  
           correct = tf.reduce\_sum(tf.to\_float(eq))  
           masked\_correct = tf.reduce\_sum(label\_mask \* tf.to\_float(eq))  
             
           return d\_loss, g\_loss, correct, masked\_correct, g\_model

## Guideline for stable Deep Convolutional GANs

* + 1. Use stride convolutions for upsampling and downsampling
    2. Use batch normalization in generator and discriminator except for the generator output layer and the discriminator input layer
    3. Use fully-connected layers only at the input of generator, which takes a uniform noise distribution, and output of discriminator
    4. Use ReLU in generator for all layers except for the output, which use tanh
    5. Use Leaky ReLU(the slope of the leak is set to 0.2) in discriminator

# Computer Vision

## Function representation of a graph

F(x, y) where x and y are the coordinates in x and y axes, respectively

## Frequency in image

A rate of change. High frequency means the intensity changes quickly from one pixel to the next

## Semantic segmentation

* + 1. Semantic segmentation doesn’t distinguish instance
    2. Upsampling
       1. Unpooling
       2. Bed of nail
       3. Max unpooling
       4. Transpose convolution

## Localization

Single object

## Object detection

* + 1. SIFT
    2. HOG
    3. Region proposal
       1. Selective search
       2. R-CNN
       3. Fast R-CNN
       4. Faster R-CNN
    4. YOLO
    5. SSD
    6. Faster R-CNN is more accurate but slower than SSD

## Instance segmentation

* + 1. Mask R-CNN

## Gradient

The slope of the tangent of the graph of function. For a graph (since a graph can be represented as a function F(x, y)), the gradient of a point can be calculated from the return of edge detection operators (such as sobel)

* + 1. Having gradients in the horizontal (G\_x) direction and vertical direction (G\_y), we can calculate the gradient magnitude, G, by  
        And the gradient direction

[atan2](https://en.wikipedia.org/wiki/Atan2) is the arctangent function with two arguments

* + 1. The gradient magnitude and directions describe the shape and patterns of intensity that make up the image

## Integral image / summed-area table

The value at each point (x, y) in the summed-area table is the sum of all the pixel above and to the left of (x, y), inclusive:

Where i(x, y) is the value of pixel at (x, y)

## Haar-like feature

Which considers adjacent rectangular regions at a specific location in a detection window, sums up the pixel intensities in each region and calculates the difference between these sums.

* + 1. As an example, below picture is a 2-rectangle feature  
        The value of this feature is calculated by: sum(black)-sum(white)

It doesn’t matter whether it’s sum(black)-sum(white) or sum(white)-sum(black), we only observe the difference between two rectangles

* + 1. Five types of Haar-like features used in Viola–Jones object detection framework  
        These features can be extended by their sizes but not by exchanging the colors of rectangles. As the above example shows, we only care the difference.

## Computer vision pipeline

* + 2. Input data
       1. In digital images, the top left coordinate is the origin
    3. Pre-processing
       1. Low-pass filter

Which can be used to preprocess image by reducing noise and unwanted traits in an image

* + - * 1. It has form of (3, 3), (5, 5), and so on.
        2. To apply the low-pass filter, we need to normalize the filter so that the sum of values in the filter is 1  
           For the above filter, we need to divide it by 9
        3. Gaussian blur

A type of low-pass filter using Gaussian function

Gaussian function in 2 dimension  
 An example a (3, 3) Gaussian blur with both standard deviation equal to 1

Larger sigma gives more weight to the neighbors. It has two influences:

I. Enhance noise reduction

II. Weaken small scale edges

* + - 1. High-pass filter

Which detects big changes in intensity or color in an image and produce an output that shows these edges

* + - * 1. It has form of (3, 3), (5, 5), and so on.
        2. Sobel Filter

Which detects abrupt intensity changes in the horizontal or vertical direction separately

Horizontal sobel filter (Sobel x):  
-1, 0, 1  
-2, 0, 2  
-1, 0, 1

Which is used to measure gradient in the horizontal direction and detect vertical edges

Vertical sobel filter (Sobel y):  
-1, -2, -1  
0, 0, 0  
1, 2, 1

Which is used to measure gradient in the vertical direction and detect horizontal edges

Two things kept in mind:  
1. The gradient direction is perpendicular to the edge direction  
2. For Sobel defined above, Sobel x detects intensity increasing in the right-direction, and Sobel y detects intensity increasing in the down-direction. These are consistent with the coordinate system used in image

Flaws:  
I. Poor localization — It triggers response in multiple adjacent pixels for ramp edges:  
 II. It responds differently for different direction edges, so it’ll miss oblique edges more than horizontal and vertical edges

* + - * 1. Laplacian

Which detects abrupt intensity changes based on the facts:

Changes are measured by 1st derivative.

The biggest change happens at the maximum or minimum of the 1st derivative, where 2nd derivative is 0

Examples of Laplacian:  
0, 1, 0  
1, -4, 1  
0, 1, 0  
Or  
1, 1, 1  
1, -8, 1  
1, 1, 1

Flaws:  
It produces spurious edges for roof edge:  
 Two edges detected, one for each side

* + - * 1. The values in the filters must add up to 0, otherwise the resulting image will be lightened or darkened based on the sum
      1. Canny Edge Detection
         1. Apply Gaussian filter to smooth the image in order to reduce the noise

It is important to understand that the selection of the size of the Gaussian kernel will affect the performance of the detector (discussion below works the same to the standard deviation, sigma). The larger the size is, the lower the detector’s sensitivity to noise. Additionally, the localization error to detect the edge will slightly increase with the increase of the Gaussian filter kernel size. A 5×5 is a good size for most cases, but this will also vary depending on specific situations.

The same discussion works for the standard deviation in the Gaussian kernel, sigma: large sigma suggests the weights of the surrounding pixels are relatively large and thus weaken the influence of the center pixel. (Also recall that, in SVM, large sigma means high bias. Both indicate that large sigma causes insensitivity to noise)

* + - * 1. Find the intensity gradients of the image
        2. Apply non-maximum suppression to get rid of spurious response to edge detection

Non-maximum suppression

An edge thinning technique. For each pixel in the gradient image, it works as follows:

1. Compare the edge strength (gradient magnitude) of the current pixel with the edge strength of the pixels in the positive and negative gradient direction

2. If the edge strength of the current pixel is the largest compared to the other two, the value will be preserved. Otherwise, the value will be suppressed (by setting it to 0)

* + - * 1. Apply double threshold to determine potential edges

Select high and low threshold values.  
Mark pixels whose gradient value is higher than the high threshold value as strong edge pixels  
Mark pixels whose gradient value is smaller than the high threshold value but larger than the low threshold value as weak edge pixels  
Suppress pixels whose gradient value is smaller than the low threshold value (by setting it to 0)

* + - * 1. Track edge by [hysteresis](https://en.wikipedia.org/wiki/Hysteresis): Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges (that is, no strong edge pixel is around the weak edge pixel).
    1. Selecting areas of interest
       1. Hough transform

A feature extraction technique, which detects shapes from a set of points. The classic hough transform was concerned with the identification of lines in images, later it has been extended to identifying positions of arbitrary shapes

* + - * 1. How does hough transform detect lines given a series of points?

Notice that a line can be represented in parametric form  
 Where the \rho is the perpendicular distance from origin to the line, \theta is the angel formed by this perpendicular line

Create an accumulator (a two-dimensional array, counting the votes in (\theta, \rho)

For each point, set \theta = 0, 1, …, 180 and calculate the \rho. For each (\theta, \rho) pair, we increment value by one in the accumulator in its corresponding (\theta, \rho) cell

Repeat above process for every point. When it’s done, we’ll obtain some local-maximum cells, each represents a detected line.

* + - * 1. Parameters for cv2.HoughLines and cv2.HoughLinesP

A binary edges

Obtained from threshold or Canny

Rho accuracy

Size of a row unit in the accumulator

Theta accuracy

Size of a column unit in the accumulator

Threshold

Minimum vote it should get to be considered as a line

minLineLength (cv2.HoughLinesP only)

Minimum length of a line.

This is not the same as the threshold: When the rho/theta accuracy is high, the counts of the accumulator would be higher than the minimum length of a line since the line drew by the threshold gets thicker

maxLineGap (cv2.HoughLinesP)

The maximum gap between two points to be considered in the same line

* + - * 1. Return values

cv2.HoughLines returns a list of rho and theta, [(rho, theta)]

cv2.HoughLinesP returns a list of endpoints of lines, [(x1, y1, x2, y2)]

* + - 1. Image segmentation

The process of partitioning a digital image into multiple segments. More precisely, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain characteristics.

* + - * 1. Image segmentation is done in two ways:  
           I. Connect a series of detecting edges  
           II. Grouping an image into separate regions by area or other traits
        2. K-means clustering

k-means clustering aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster.

Randomly choose k points (aka, center points) in data space so as to partition the space into k clusters

The data space, for an image, is RGB color space

Assign every data point to a cluster, based on nearest center points

Update center points to the mean of values in their own cluster

That’s the reason why we initially call these points center points.

Repeat ii and iii until convergence is reached

The convergence is defined by us, it can be simply some number of iterations, or more technically, based on how much the center points move after each iteration, or

* + 1. Feature extraction
       1. How do we distinguish different objects?
          1. We look for specific features that are unique, trackable, and comparable
       2. Types of features
          1. Edge

Area with a high intensity gradient

* + - * 1. Corner

The intersection of two edges

* + - * 1. Blob

Region-based feature. Area of extreme brightness or unique texture

* + - 1. Corner detector
         1. How to detect a corner?

Take a window, and shift the window around an area in an image

For a corner, shifting the window in any direction should yield is a big variation in the direction and magnitude of the gradient

Moreover, for a edge, there should be no variation when shifting the window along the edge direction.

Harris corner detector

Theoretical basis

Considering taking a window (a image patch) of size (x, y) and shifting it by (Delta(x), Delta(y)), the weighted sum of squared differences between these two windows, S(x, y), is given by:  
 Where w is the window function which gives weights to pixels underneath, I is the intensity function. Moreover, the sum of squared difference (i.e. I(u+x, v+y) - I(u, v)) should be large for distinctive patches, very small for constant patches.

The window function is either a rectangle widow or Gaussian window as shown below

I(u+x, v+y) can be approximated by a Tayler expansion  
 Now we have

Taylor expansion for 2D functions:

Above approximation only takes the part where n=1

It can be shown that  
 Where

I{x} and I{y} can be easily calculated by Sobel

M has two eigenvalues, \lambda{1}, \lambda{2}. Based on the magnitude of the eigenvalues, there are three cases  
1. Both are small, the region is flat  
2. One is much larger than the other, an edge is detected  
3. Both are large and one is approximate to the other, a corner is detected

Note, although the matrix part is singular, the sum of singular matrices is not necessarily singular, so M has two eigenvalues

The computation of eigenvalues is computationally expensive, we use following function R, instead  
 Where det(M) is the determinant, trace(M) is the trace of M, i.e. \lambda{1} + \lambda{2}, k is a tunable sensitivity parameter, usually in range of [0.04, 0.06]

Now the cases become  
|R| is small for a flat region  
R is negative with large magnitude for an edge  
R is large for a corner

Process of Harris corner detection algorithm

Color to grayscale

Spatial derivative calculation

Calculate I{x}, and I{y}

Structure tensor setup

Calculate the structure tensor M

Harris response calculation

Calculate R

Thresholding

Define a threshold, suppress those whose R is less than the threshold

* + - 1. Feature vector

An n-dimensional vector of numerical features that represent some object

* + - 1. Histogram of Oriented Gradients (HOG)

An feature descriptor used in computer vision, which counts occurrences of gradient orientation in localized portions of an image

* + - * 1. Histogram

A graph which gives you an overall idea about the intensity distribution of an image

Bin

A subdivision which contains a specified range of values

Why do we use bin?

Bins reduce the dimensionality of data, they represent image data in a more compact way

Group data into ranges allows for more flexibility (prevent overfitting) in identifying similar objects based on their feature vectors.

Dimensionality

The number of parameters for which we collect the data

Range

The range of intensity values we want to measure

* + - * 1. How does HOG work?

Calculate magnitude and direction of the gradient at each pixel

Group these pixels into square cells

Counts how many gradients in each cell fall in a certain range of orientations, and place all the directional data in a histogram (one cell for one histogram). For example

These histograms are actually feature vectors

Then it uses a block — which contains many cells — to shift around the image just as the kernel in CNN. At each step, it normalizes the histograms in it and produces feature vectors from those histogram. This suggests that each histogram may be used as feature vectors more than once, but since the normalizer changes when the block shifts, the output feature vector from a histogram won’t stay the same.

Use these HOG feature vectors to train a model and classify objects

The idea is, among images of the same object at different scales or orientations these HOG feature can be used to detect the object whenever and however it appears