- **Types of ML**: unsupervised(learn a model of data), supervised(learn mapping from input to target), reinforcement(learn from mistakes), imitation(type of reinforcement)
- **Linear Regression**: supervised, $X^{N,1+D}W^{1+D,1} = \boldsymbol{y}^{\boldsymbol{N,1}}$
  - Minimize SSE(sum squared error)
  - With SSE, $W = (X^TX)^{-1}X^Tt$
- **Gradient Descent**
  - $w(t+1) = w(t) - \alpha(\partial J / \partial w)$
  - Batch descent: $w_i = w_i + \alpha\frac{1}{N}\sum_{n=1}^{N}\delta^n x_i^n$
  - SGD: update weight 1by1, needs shuffle
- **Perceptron**
  - Outputs 1 or 0
  - $w_i = w_i + \alpha(t^n - y^n)x_i^n$ where $\delta^n = (t^n - y^n)$
  - Can learn whatever it can computes, but slow
- **Maximum Likelihood Objective Function**
  - Adjust your parameters to maximize likelihood of data distribution you see
  - SSE assumes Gaussian distribution and cross entropy assumes Bernoulli/multinomial distribution
  - We can cluster datapoints by pushing same category together and pushing different one apart—supervised clustering(we know data's category)
    - Siamese networks: two identical NN
    - Loss function: $L(W, Y, \vec{X}_1, \vec{X}_2) = (1-Y)\frac{1}{2}(Dw)^2 + (Y)\frac{1}{2}\{\max(0, m - Dw)\}^2$ where D is distance between networks' output, m is margin, Y is indicator for different pair
- **Backprop**
  - $w_{ij} = w_{ij} + \alpha\delta_j z_i$ where $\delta_j = t_j - y_j$ for output unit and $g'(a_j)\sum_k \delta_k w_{jk}$ for hidden unit
- **Generalization**
  - More data/augment existing data
  - Regularization: minimize $J = E + \lambda C$ to penalize too complex model
    - $C = |W|$(L1);$C = ||W||_2^2$(L2)
  - Dropout: randomly turn off hidden units when training
  - Early Stopping: leave a holdout set from training data, watch the error on it and stop training if it starts to rise
  - Add Gaussian noise into inputs/model/outputs
- **Tricks for Training**
  - SGD and minibatch: faster to converge, better generalization, adaptive, take advantage of dataset redundancy, need shuffling
  - Normalize data
    - All positive inputs make weight changes all + or -
    - Correlated inputs: redundant information
    - Different scale: largest weight change for large inputs, bad if two inputs are equally important
  - PCA: shifts mean of input var to 0, decorrelates inputs, dimensionality reduction
  - Z-scoring: shift mean to 0, make variables same size, no decorrelate, no dimension reduction
  - Activation function
    - Sigmoid: all positive output, bad. Use $f(x) = 1.7159\tanh(0.667x)$ instead
    - ReLU: $\max(0, x)$ only non-negative output, input issue not as big of concern, also can use batch normalization
  - Weight initialization
    - Init 0, if sigmoid, delta of hidden will be the same, hidden units all compute same feature. If tanh, all weights stay zero.
    - Want weighted sum of inputs to be 0 mean and unit deviation (linear range of sigmoid) because gradients will be largest and network learn any linear part before non-linear part
    - sd(aj) = sqrt(var(XW)) = sqrt(var(W)) = sqrt(sum_i(wij)^2)
    - Fan-in: number of inputs to unit j, fan-out: number of outputs
    - Let fan-in be m, init weights to mean 0, sd 1/(m)^½
    - ReLU

Xavier initialization (Glorot & Bengio, 2010):

– Where n_j and n_{j+1} are the fan-in and fan-out of a unit, respectively)

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right] \quad (16)$$

Better: Kaiming initialization (He, et al., 2015):

$W \sim \mathcal{N}(0, \sqrt{\frac{2}{n_l}})$, where n_l is the fan-in, and

- Here, $\mathcal{N}$ denotes a normal distributio

  - Batch normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

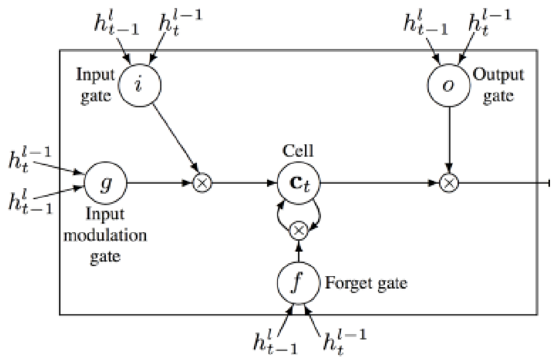
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

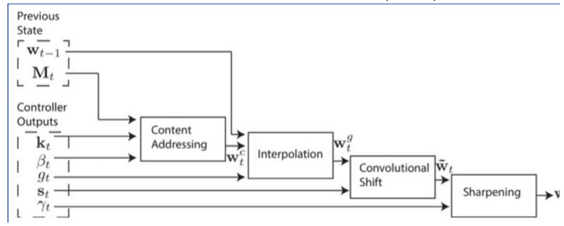
    - Treated as a layer between output of previous layer the input of next layer
    - Operation is differentiable so can backprop
  - Convergence with a multi-layer, non-linear network indicates that the network has found a local minima along the error surface
  - Momentum
    - Motivation: – Move quickly in directions with small but consistent gradients. – Move slowly in directions with big but inconsistent gradients.
    - Nesterov: update w with accumulated gradient ,calculate new gradient then make correction
  - Adaptive learning rate
    - rprop for full batch and rmsprop for minibatch
- **Convnets**
  - Why convnet: images are huge!
  - Convnets use four principles:
    - Pixels depend on nearby pixels (locality): Hence, small receptive field
    - The statistics of visual inputs are invariant across images (stationary statistics): Hence, replicate receptive fields across images
    - Objects don't change identity based on location: (translation invariance): Spatial pooling
    - Objects are made of parts(compositionality): receptive fields get larger deeper in the net
  - Components: convolution(kernel dot inputs)--nonlinearity(rectified linear)--pooling
    - Convolute filter (3x3) computed on whole image(*stationary statistics*)
    - Nonlinearity: applied per pixel (ReLU)
    - Spatial Pooling: max is best, avg/sum less. Pooling provides larger receptive field and invariance to small transformations
    - Batch normalization: normalize layer inputs to eliminate covariance shift caused by changing previous layer parameters
  - Deep network:
    - Deeper is better: more abstract feature, larger reception field
    - To pass gradients effectively: deep supervision, skip connections, etc.
    - Reuse pretrained networks and modify for new tasks
  - Translation invariance(good, built into the network); scale invariance(good, learned from data, can be built in); rotational invariance(bad, can be built in)
  - Dilation: increase receptive field without losing resolution. Pooling and stride also increase receptive field but reduce resolution
  - Network in Network: 1x1 kernel to add nonlinearity
  - Global Average Pooling: use feature map's average as output instead of softmax. Can visualize activation to see what network is looking at
  - GoogleNet: multiple kernel sizes at same layer: look at different scales; 1x1 kernel for reducing dimension and number of computation
  - ResNet: introduce pass through between layers to help gradient propagation
  - Visualize Features: deconvolution(zero out all features except the max and propagate back); gradient descent back(optimize input to maximize particular output by taking gradient of output and backprop)
  - Adversarial: calculate gradient at the output and climb up
  - Training tips: use SGD/mini-batch; Adam optimizer is popular; start with big learning rate and anneal it; use Nesterov momentum; examine the data and normalization; measure both training and validation error; test with small dataset before running in full.
  - Signs of good training: hidden units are sparse across samples and across features; learned filters exhibit structure and are uncorrelated.
  - Problems & Fix:
    - Training diverges: learning rate too high
    - Bad accuracy: make the network bigger/deeper; visualize feature and fix optimization
    - Training is slow: use matrix operation not for loop

- **Map time into:**
  - Space: NETTalk, Transformer
    - Autoregressive models Predict the next term in a sequence from a fixed number of previous terms using "delay taps".
    - Feed-forward models generalize autoregressive models by using 1+ layers of hidden units
    - Work well for small problems
    - Difficulty dealing with variable input sizes
    - But transformer changed all this
  - State: RNN, LSTM
    - Use activation memory to process input based on previous ones
- **RNN**
  - Recurrent network can be unrolled in time and becomes feed-forward
  - BPTT: compute gradient, propagate backward in time, and compute average change between time steps for a weight
  - Problem of BPTT: backprop is linear, so gradient can vanish/explode. Hard to train long network(solved by LSTM)
- **LSTM**
  - Three gates: write(input), keep(info stay in the cell through time), read(output)

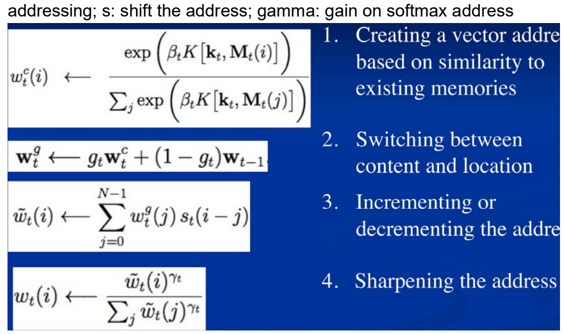


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} T_{2n,4n} \begin{pmatrix} \mathbf{D}(h_t^{l-1}) \\ h_{t-1}^{l} \end{pmatrix}$$
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

- **RNN generation**
  - N to 1: sentient classification; 1 to N: image captioning; N to N(after): machine translation; N to N(sync): label frames of video
  - Feed previous output into input
  - Combined with conv to generate image caption, can include original feature to overcome longer timesteps
- **Transformers**
  - Feedforward, same weights for all inputs, parallelizable: fast to train
  - A tower of encoder/decoder contains feedforward layers and attention
  - Encoder has self-attention: it can look at multiple inputs
  - Attention: tower communicate with other towers through attention network
    - KQV: key(what I have), query(what I'm looking for), value(what I give when other towers ask)
    - Output: Softmax(KQ)*V (softmax of all towers' key times tower's own query forms interest vector, then times all towers' value)
  - Image transformers: same architecture(encoders), no convolution, taking patches of image with positional encodingResidual: preserve gradient
- **Neural Turing Machine**
  - Neural net can learn to program but slow and difficult to adapt
  - NTM: add a structured memory to neural controller to read and write
  - Neural controller can be feedforward or recurrent(better)



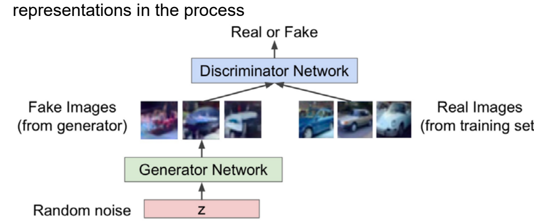
  - K: key; beta: gain on content match; g: switch between content/location addressing; s: shift the address; gamma: gain on softmax address



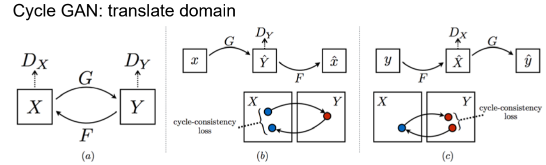
  - Can learn to program: Copy, repeated copy, associative recall, priority sort
- **Reinforcement Learning**
  - An agent act in an environment, which changes states and gives the agent an reward. Goal is to maximize reward
  - Agent learns a policy: given state, output a probability function of actions
  - Policy gradient: sample the softmax policy distribution at each step; use the sample as teacher; compute the weight change and keep running average; multiply the weight changes with reward sign when game ends; update the network
  - Discount rate: exponentially reduce possible future rewards(0-1, the bigger the more far-sighted)
  - Markov property: all the agent needs to know at any time is the current state(which usually isn't true)
  - Model based learning: the agent either has available to it, or it learns, a model of its environment. Most RLs are model-free
  - Model: probability distribution of possible outcome states given current state and actions
  - Value of state: expected reward in this state given policy
  - Q learning: update current state's Q value based on Q values of states you get to. Overtime the Q values approximate final reward
  - Minimax: a player maximizing own reward while the opponent minimizing its reward. Tree's branching factor equals to number of legal moves, cannot use for larger games
  - Temporal Difference Gammon

$$w_{t+1} - w_t = \alpha(Y_{t+1} - Y_t)\sum_{k=1}^{t} \lambda^{t-k} \nabla_w Y_k$$

    - The weights were updated using a temporal difference rule on every move: That is, the current estimate of the board value was updated to be closer to the next board's value
    - lambda: how far error feeds back(stop when 0, arbitrarily far when 1)
- **Alpha-go:**
  - Two policy networks with supervised training. Shallow for rollout, deep for training: play against younger selves with reinforcement learning
  - A value network predicts winner given states: supervised training
  - Monte Carlo Tree Search:
    - Traverse the tree with depth L
    - Selection actions based on Q+upper confidence bound
    - The upper confidence bound increases as a move is not tried
    - Expand the tree based on selection
    - Evaluate nodes by average of shallow value network and value network, propagate value up the tree
    - Back up the tree for future use
- **Alpha-zero:**
  - One network for both value and policy
  - Simplified MCTS(selection, expand & evaluate, backup)
  - Use resnet with BN
- **GAN**
  - Unsupervised learning to generate data from a distribution
  - Recall autoencoder, which is trained to replicate the input and learns representations in the process



  - GAN has only decoder(generator) and an adversary(discriminator)
  - Discriminator learns to identify fake image from real, and generator tries to fool the discriminator
  - Discriminator trains by supervised label, generator trains by gradient ascent on discriminator
  - Conditional GAN: feed generator with images instead of noise
  - Cycle GAN: translate domain



  - ReCycle GAN: map videos to videos
    - Use Unet-based predictor to predict next frame in target domain, then translate back into original domain
- **SimCLR: self supervision**
  - In a batch, augment a data to get positive pairs and use other images in the batch as negative pairs
  - Add an extra layer on final output and minimize its outputs on positive pairs. Then throw away the final layer after training is complete
    - Rationale: by mapping augmented data exactly the same, some information is lost
  - Use normalize temperature-scales cross entropy loss
    - Minimize distance of positive pair and vice versa
  - Randomized data augmentation