Machine Perception

Doruk Çetin

September 13, 2019

Contents

1 Deep Learning Basics

Classification, loss functions, image features, traditional ML vs deep learning, representation learning, perceptron learning algorithm, sigmoid function, logistic regression.

Some basic equations:

- Quotient rule: $f(x) = \frac{g(x)}{h(x)} \implies f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{[h(x)]^2}$
- Hinge loss: $\max\{0, 1 - y \cdot \hat{y}\}$
- $\frac{\partial \|x\|_1}{\partial x} = \text{sgn}(x) = (-1)^{\mathbb{1}[x<0]}$
- Binary cross-entropy: $-y \log \hat{y} - (1-y) \log(1-\hat{y})$
- Multi-class cross-entropy: $-\sum_{k=1}^{K} y_{i,k} \log(\hat{y}_{i,k})$

2 Training Neural Networks

- Maximum likelihood estimation (MLE): minimizing negative log-likelihood
- The activation function should be non-linear, or the resulting MLP is an affine mapping with a peculiar parametrization.
- Universal approximation theorem: a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of R^n, under mild assumptions on the activation function. Although feed-forward networks with a single hidden layer are universal approximators, the width of such networks has to be exponentially large.
- Chain rule:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

Pitfall: all zero initialization. This turns out to be a mistake, because if every neuron in the network computes the same output, then they will also all compute the same gradients during backpropagation and undergo the exact same parameter updates. In other words, there is no source of asymmetry between neurons if their weights are initialized to be the same.

Warning: small random numbers. It's not necessarily the case that smaller numbers will work strictly better. For example, a Neural Network layer that has very small weights will during backpropagation compute very small gradients on its data (since this gradient is proportional to the value of the weights). This could greatly diminish the "gradient signal" flowing backward through a network, and could become a concern for deep networks.

Xavier initialization: has variance inversely proportional to fan-in n_{in} (previous layer size) and fan-out n_{out} (next layer size): $\text{Var}(W_i) = \frac{2}{n_{in} + n_{out}}$

Parameter updates

- Vanilla update: change parameters along the negative gradient

- Momentum update: gradients act as a force and change the velocity, indirectly affecting the position. The parameter vector will build up velocity in any direction that has consistent gradient.
- Nesterov momentum: compute gradient at the future approximation ("lookahead") position, which is the current position plus the momentum term.
- Adagrad: keeps track of per-parameter sum of squared gradients, which is then used to normalize the parameter update step, element-wise. Weights that receive high gradients will have their effective learning rate reduced, while weights that receive small or infrequent updates will have their effective learning rate increased. A downside of Adagrad is that in case of Deep Learning, the monotonic learning rate usually proves too aggressive and stops learning too early.
- RMSProp: modifies Adagrad by using a moving average of squared gradients instead.
- Adam: update looks exactly as RMSProp update, except the "smooth" version of the gradient is used instead of the raw (and perhaps noisy) gradient vector.

2.1 Activation Functions

- Sigmoid:
$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}, \text{ with } \sigma'(x) = \sigma(x)(1-\sigma(x))$$
- tanh:
$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \text{ with } \tanh'(x) = 1 - \tanh^2(x)$$
- Softmax:
$$\text{softmax}(\nu_i) = \frac{\exp(\nu_i)}{\sum_j \exp(\nu_j)}$$
- Rectified linear unit:
$$ReLU(x) = \max(0, x), \text{ with } ReLU'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$
- Leaky ReLU:
$$LeakyReLU(x) = 1(x < 0)(\alpha x) + 1(x \geq 0)(x)$$

Sigmoids saturate and kill gradients: A very undesirable property of the sigmoid neuron is that when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. Additionally, one must pay extra caution when initializing the weights of sigmoid neurons to prevent saturation. For example, if the initial weights are too large then most neurons would become saturated and the network will barely learn.

Sigmoid outputs are not zero-centered: This is undesirable since neurons in later layers would be receiving data that is not zero-centered. This has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive (e.g. $x > 0$ elementwise in $f = w^T x + b$), then the gradient on the weights w will during

backpropagation become either all be positive, or all negative (depending on the gradient of the whole expression f). This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights. However, notice that once these gradients are added up across a batch of data the final update for the weights can have variable signs, somewhat mitigating this issue.

tanh vs sigmoid: Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centered. Therefore, in practice the *tanh non-linearity is always preferred to the sigmoid nonlinearity.* Also note that the tanh neuron is simply a scaled sigmoid neuron, in particular the following holds: $tanh(X) = 2\sigma(2x) - 1$.

ReLU variants: Some people report success with this form of activation function (Leaky ReLU), but the results are not always consistent. The slope in the negative region can also be made into a parameter of each neuron, as seen in PReLU neurons. However, the consistency of the benefit across tasks is presently unclear.

Advantages and disadvantages of ReLU:

+ It was found to greatly accelerate (e.g. a factor of 6 in Krizhevsky et al.) the convergence of SGD compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form.
+ Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.
− Unfortunately, ReLU units can be fragile during training and can "die". For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold. For example, you may find that as much as 40% of your network can be "dead" (i.e. neurons that never activate across the entire training dataset) if the learning rate is set too high. With a proper setting of the learning rate this is less frequently an issue.

2.2 Backpropagation

The derivative on each variable tells you the sensitivity of the whole expression on its value.

Importance of staged computation for practical implementations: You always want to break up your function into modules for which you can easily derive local gradients, and then chain them with chain rule.

- Explicitly: $a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$
- Vectorized: $a^l = \sigma(w^l a^{l-1} + b^l)$

Gradient checking: Simply compare the analytic gradient (centered formula) to the numerical gradient:

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x-h)}{2h}$$

Node intuitions:

- + distributes the upstream gradient to each summand
- max routes the upstream gradient
- × switches the upstream gradient

2.3 Regularization methods

Regularization interpretation: The regularization loss in both SVM/Softmax cases could in this biological view be interpreted as gradual forgetting, since it would have the effect of driving all synaptic weights w towards zero after every parameter update

- Data standardization: some machine learning models assume zero mean, unit variance. Standardize dataset and use same mean and standard deviation for the test data set.
- Data augmentation: expand data set by transforming existing samples to produce new samples, need to ensure consistency between transformed data sample and label. For classification introduces invariance, for regression introduces equivariance.
- Noise injection: introduce robustness to perturbation.
- Norm penalties: penalize a norm of the weights.
- Early stopping: no guarantee that i^* remains optimal for the entire training dataset, either retrain on entire data set for i^* iterations, or continue training from previous found parameters.

Dropout:

- Prevents complex co-adaptations in which a feature detector is only helpful in the context of several other specific feature detectors. So a hidden unit cannot rely on other hidden units being present. Another way to view the dropout procedure is as a very efficient way of performing model averaging with neural networks.
- During test time: rescale weights instead of summing over exponentially many weights. This is far more efficient than averaging the predictions of many separate models. We can also rather invert the dropout during the training phase.
- Models share weight in dropout, whereas bagging has independent models
- Dropout trains only a small percentage of its models, whereas bagging trains all models until convergence

Batch normalization:

- Internal covariance shift: the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities.
- Standardize input distribution to each layer, by learning a linear transform
- Original paper: The goal of Batch Normalization is to achieve a stable distribution of activation values throughout training, and in our experiments we apply it before the nonlinearity since that is where matching the first and second moments is more likely to result in a stable distribution.

Semi-supervised learning by combining an autoencoder with a classifier: autoencoder is only there to provide signal to shared weights, it is irrelevant at the end. Labeled data is fed to both ends but unlabeled data is only fed to autoencoder.

Multi-task learning: divide model into shared and task-specific parameters. Multi-task learning happens concurrently whereas in transfer learning we adapt the model from one task to another, i.e. cutting the top layers to get a learned representation.

There are better ways of doing semi-supervised and multi-task learning than described approaches. Final notes:

- Can use combinations of these
- Not all guaranteed to work, but some will most probably
- Just try and see, many are cheap to implement
- If no improvement after small number of epochs, discard

3 Convolutional Neural Networks

Given a transformation T:

- T is linear if $T(\alpha u + \beta\nu) = \alpha T(u) + \beta T(\nu)$
- T is invariant to f if $T(f(u)) = T(u)$
- T is equivariant to f if $T(f(u)) = f(T(u))$

Any linear, shift-equivariant function can be written as a convolution

- Correlation (e.g. template matching):

$$I'(i,j) = \sum_{m=-k}^{k} \sum_{n=-k}^{k} K(m,n) I(i+m, j+n)$$

- Convolution (e.g. point spread function):

$$\begin{aligned} I'(i,j) &= \sum_{m=-k}^{k} \sum_{n=-k}^{k} K(m,n) I(i-m, j-n) \\ &= \sum_{m=-k}^{k} \sum_{n=-k}^{k} K(-m,-n) I(i+m, j+n) \end{aligned}$$

- So if $K(i,j) = K(-i,-j)$ then correlation becomes equal to convolution

CNNs are very effective models thanks to weight-sharing and repetition of convolution operation, which also provides invariance to certain image operations such as translation and rotation. The **parameter space** can be reduced considerably in comparison to fully connected layers. Choosing huge filter sizes goes against this intuition, and in recent research it was shown, that instead of increasing filter sizes, creating **deeper models** is generally a better idea.

3.1 Layers and Architectures

The **receptive field** is defined as the region in the input space that a particular CNN's feature is looking at (i.e. be affected by).

Dilations: In dilated convolutions, sometimes also called "à trous", we introduce holes in the filter, i.e. we spread the filter over a wider area but without considering some pixels inside that area in the computation of the dot product. This allows for a faster growth of the receptive field in deeper layers than with standard convolutions. The intuition behind is that it is easier to integrate global context into the convolution operation.

Pooling Layers: Downsampling or pooling layers concentrate the information so that deeper layers focus more on abstract/high-level patterns. You can apply strided convolutions to apply downsampling. A decreased image size also speeds up the processing time in general because less convolutions are necessary on subsequent layers. Furthermore, pooling allows for some translation invariance on the input. A common choice is max-pooling, where only the maximum value occurring in a certain region is propagated to the output.

Activation before/after pooling: The theory is to use the order Convolutional Layer - Non-linear Activation - Pooling Layer. In case of max-pooling layer and ReLU the order does not matter (both calculate the same thing): since $relu(max\_pool(x)) = max\_pool(relu(x))$ we can save 75% of the relu-operations by max-pooling first. Sadly this optimization is negligible for CNN, because majority of the time is used in convolutional layers. The same thing happens for almost every activation function (most of them are non-decreasing). But does not work for a general pooling layer (average-pooling).

Odd-sized filters: If you want to think of the convolution as a filter with some response, with an odd size you can "center" that filter on a pixel in your input. It also gets ugly handling edge effects if your kernel isn't odd. Also even kernels can't have horizontal/vertical symmetry until they get pretty large.

CNN architectures:

- VGG: more depth, large receptive fields, fewer parameters
- GoogLeNet: Inception module, no FC layers, dimensionality reduction via 1x1 convolutions, auxiliary classification heads
- ResNet: ultra-deep, faster than VGG during runtime (despite 8x more layers), residual connections
- DenseNet, FractalNet, Wide ResNet, ResNetX, ...

Fully Convolutional Networks: idea is to downsample and upsample inside the network, as the convolutions at original resolution are very expensive.

- Downsampling: pooling, strided convolution
- Upsampling: unpooling (nearest neighbor, bed of nails, max unpooling -remembering element in which position was the max), strided transpose convolution (learnable upsamplin, upconvolution, fractionally strided convolution, backward strided convolution, deconvolution -bad name)

3.2 Backpropagation in CNNs

Backward pass in convolutional layers is again a convolution, where forward pass is $Y = X * K$ (apply zero padding to match the output dimensions):

- Backpropagating the signal:

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} * Rot_{180}(K) = \frac{\partial E}{\partial Y} \star K$$

- Parameter update:

$$\frac{\partial E}{\partial K} = \frac{\partial E}{\partial Y} * Rot_{180}(X) = \frac{\partial E}{\partial Y} \star X = Rot_{180}\left(X \star \frac{\partial E}{\partial Y}\right)$$

Change of variables to match the layer-wise notation:

- $X = z^{(l-1)}$
- $\frac{\partial E}{\partial X} = \delta^{(l-1)}$
- $Y = z^{(l)}$
- $\frac{\partial E}{\partial Y} = \delta^{(l)}$
- $K = w^{(l)}$
- $\frac{\partial E}{\partial K} = \frac{\partial E}{\partial w^{(l)}}$

4 Recurrent Neural Networks

We can model a dynamic system as $s^t = f(s^{t-1}; \theta)$, where state at time t (s^t) depends on a function of s^{t-1} parametrized by θ. To allow for a more expressive model we can incorporate inputs for each time point of the sequence and model the state as $s^t = f(s^{t-1}, x^t; \theta)$

Representing recurrence by folding/unfolding has two main advantages:

- Learned model always has the same input size. Its specified in terms of transition from one state to another state. (Not variable length history of states)
- It is possible to use the same transition function f with the same parameters at every time step.

In **(Vanilla) RNNs**, the state consists of a single hidden vector $h^t = f(h^{t-1}, x^t, W)$, such that

- $h^t = \tanh(W_{hh}h^{t-1} + W_{xh}x^t)$
- $y^t = W_{hy}h^t$

In matrix form, vanilla RNN is

$$h_t^l = \tanh W^l \begin{bmatrix} h_t^{l-1} \\ h_{t-1}^l \end{bmatrix}$$

Recurrent neural networks offer a lot of flexibility:

- One-to-one: vanilla neural networks
- One-to-many: e.g. image captioning
- Many-to-one: e.g. sentiment classification
- Many-to-many: e.g. machine translation
- Synced many-to-many: e.g. frame-level video classification

4.1 Backpropagation through time (BPTT)

Intuition: treat unrolled recurrent model as multi layer network (with unbounded number of layers) and perform backprop.

Non-truncated BPTT: it is very slow and expensive to backpropagate the error so many steps, also the gradients may blow up while backpropagating through the entire sequence.

Truncated BPTT: (from Sutskever)processes the sequence one timestep at a time, and every k_1 timesteps, it runs BPTT for k_2 timesteps. Tensorflow, for example, uses $k_1 = k_2$ (this approach is also called epochwise truncated BPTT).

4.2 Vanishing gradients problem

The problem with long sequences is due to the fact that the model basically goes through the same weight matrix (W_{hh}) that gets copied over the timesteps many, many times.

The fundamental problem here isn't so much the vanishing gradient problem or the exploding gradient problem. It's that the gradient in early layers is the product of terms from all the later layers. When there are many layers, that's an intrinsically unstable situation. [...] As a result, if we use standard gradient-based learning techniques, different layers in the network will tend to learn at wildly different speeds.

As shown in the Pascanu paper, if the dominant eigenvalue of the matrix W_R is greater than 1, the gradient explodes. If it is less than 1, the gradient vanishes. [link]

Gradient clipping may act as a simple workaround for the exploding gradients problem.

4.3 Long Short Term Memory (LSTM)

Gates: Cell state: Output:

$$\begin{bmatrix} i \\ f \\ o \\ g \end{bmatrix} = \begin{bmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{bmatrix} W^l \begin{bmatrix} h_t^{l-1} \\ h_{t-1}^l \end{bmatrix} \qquad c_t^l = f \odot c_{t-1}^l + i \odot g \qquad h_t^l = o \odot \tanh c_t^l$$

LSTMs solve the problem by creating a connection between the forget gate activations and the gradients computation, this connection creates a path for information flow through the forget gate for information the LSTM should not forget. This might all seem magical, but it really is just the result of two main things:

- The additive update function for the cell state gives a derivative thats much more "well behaved".
- The gating functions allow the network to decide how much the gradient vanishes, and can take on different values at each time step. The values that they take on are learned functions of the current input and hidden state.

Some **LSTM variants**:

- Peephole connections: We let the gate layers (either all or some) look at the cell state.

- Coupled forget and input: Instead of separately deciding what to forget and what we should add new information to, we make those decisions together. We only forget when we're going to input something in its place. We only input new values to the state when we forget something older.
- Gated Recurrent Unit (GRU): It combines the forget and input gates into a single "update gate." It also merges the cell state and hidden state, and makes some other changes.
- Many other, e.g. Clockwork RNNs, Depth Gated RNNs

5 Deep Generative Models

Generative Modelling: Given training data, generate new samples, drawn from "same" distribution. Most generative models either explicitly or implicitly reason about the underlying pdf and try to fit the model to it (explicit) or try to capture what pdf expresses (implicit).

Taxonomy of generative models:

- Explicit density
  - Approximate
    - Variational: VAE
    - Markov Chain: Boltzmann Machines
  - Tractable: Belief Nets, NADE, MADE, PixelRNN/CNN
- Implicit density
  - Direct: GAN
  - Markov Chain: GSN

5.1 Variational Autoencoders

Autoencoders: Finding "meaningful degrees of freedom" that describe the high dimensional signal with lesser dimensions. An autoencoder consists of an encoder (f) and a decoder (g). A simple example of such an autoencoder would be with both f and g linear, in which case the optimal solution is given by PCA.

Reconstruction loss encourages things that are similar to be close to each other in latent space, which creates a lack of support in latent space for things in between (you can't generate novel samples). What we would really like is a compact, Gaussian-like latent space.

Variational autoencoders (VAEs) are, in a nutshell, probabilistic versions of autoencoders. Often prior is assumed to be Gaussian for mathematical convenience (closed form solutions) and the conditional is modelled via a neural network. Data likelihood becomes:

$$p_\theta(\mathbf{x}) = \int_z p_\theta(\mathbf{x} \mid \mathbf{z}) p_\theta(\mathbf{z}) dz$$

integral over continuous random variable makes the likelihood intractable. Solution is to define an additional network $q_\theta(\mathbf{z} \mid \mathbf{x})$ to approximate $p_\theta(\mathbf{z} \mid \mathbf{x})$

In practice, most people ignore the probabilistic part of the decoder and draw a single sample, taking directly the mean.

5.1.1 Data log-likelihood

Data likelihood is $p_\theta(x) = \int_z p_\theta(x \mid z) p_\theta(z) dz$, where

- $p_\theta(z)$ is a simple prior. It is usually Gaussian (an isotropic normal distribution) and it controls the latent factors. It is tractable.
- $p_\theta(x \mid z)$ is the conditional. It is usually complex since it needs to generate outputs. It is parametrized via decoder neural network. It is tractable.
- $\int_z$ is an integral over continuous variables, making it intractable. Since the data likelihood is intractable, posterior is also intractable as follows:

$$p_\theta(z \mid x) = \frac{p_\theta(x \mid z) p_\theta(z)}{p_\theta(x)}$$

- Solution is to approximate posterior $p_\theta(z \mid x)$ through an encoder network $q_\phi(z \mid x)$

Here we disentangle the log-likelihood to obtain a lower bound:

$$\begin{aligned}
\log p_\theta(x^{(i)}) &= \mathbb{E}_{z \sim q_\phi(z|x^{(i)})} \left[\log(p_\theta(x^{(i)})) \right] && \text{(Doesn't depend on } z\text{)} \\
&= \mathbb{E}_z \left[\log \frac{p_\theta(x^{(i)} \mid z) p_\theta(z)}{p_\theta(z \mid x^{(i)})} \right] && \text{(Bayes' rule)} \\
&= \mathbb{E}_z \left[\log \frac{p_\theta(x^{(i)} \mid z) p_\theta(z)}{p_\theta(z \mid x^{(i)})} \frac{q_\phi(z \mid x^{(i)})}{q_\phi(z \mid x^{(i)})} \right] && \text{(Mult. by constant)} \\
&= \mathbb{E}_z \left[\log p_\theta(x^{(i)} \mid z) \right] - \mathbb{E}_z \left[\log \frac{q_\phi(z \mid x^{(i)})}{p_\theta(z)} \right] + \mathbb{E}_z \left[\log \frac{q_\phi(z \mid x^{(i)})}{p_\theta(z \mid x^{(i)})} \right] && \text{(Logarithms)}
\end{aligned}$$

$$= \mathbb{E}_z \left[\log p_\theta(x^{(i)} \mid z) \right] - D_{KL}(q_\phi(z \mid x^{(i)}) \parallel p_\theta(z)) + D_{KL}(q_\phi(z \mid x^{(i)}) \parallel p_\theta(z \mid x^{(i)}))$$

This is the result:

- $\mathbb{E}_z \left[\log p_\theta(x^{(i)} \mid z) \right]$ is the reconstruction loss, tractable through sampling.
- $D_{KL}(q_\phi(z \mid x^{(i)}) \parallel p_\theta(z))$ makes approximate posterior as similar as to prior. Tractable in closed form if prior is Gaussian, for example.
- $D_{KL}(q_\phi(z \mid x^{(i)}) \parallel p_\theta(z \mid x^{(i)})) \geq 0$ is intractable.
- $\mathcal{L}(x^{(i)}; \theta, \phi) = \mathbb{E}_z \left[\log p_\theta(x^{(i)} \mid z) \right] - D_{KL}(q_\phi(z \mid x^{(i)}) \parallel p_\theta(z))$ form the variational **ELBO lower bound**, data is at least as likely as $\mathcal{L}$.
- Training is $\theta^*, \phi^* = \arg\max \sum_1^N \mathcal{L}(x^{(i)}; \theta, \phi)$
- Shorthand for the lower bound: $\mathbb{E}_{z \sim q_\phi} \left[\log \frac{p_\theta(x,z)}{q_\phi(z|x)} \right]$

Gradient computation for backprop (where w is the deterministic NN):

$$\begin{aligned}\nabla_{\theta,\phi}\,\mathbb{E}_{z\sim q_\phi}\left[\log\frac{p_\theta(x,z)}{q_\phi(z\mid x)}\right] &= \nabla_{\theta,\phi}\,\mathbb{E}_{\epsilon\sim\mathcal{N}(0,1)}\left[\log\frac{p_\theta(x,f(x,\epsilon,\theta))}{q_\phi(f(x,\epsilon,\phi)\mid x)}\right]\\ &= \mathbb{E}_\epsilon\left[\nabla_{\theta,\phi}\log\frac{p_\theta(x,f)}{q_\phi(f\mid x)}\right]\\ &\simeq \frac{1}{k}\sum_{i=1}^{k}\nabla_{\theta,\phi}\log w(x,f,\theta,\phi)\end{aligned}$$

5.1.2 Limitations of VAEs

VAEs have a tendency to produce very blurry images. This comes from being forced to estimate the covariance of the entire model distribution by optimizing a per-sample loss. Basically, we cannot really differentiate the high-frequency and low-frequency details, typically producing overly smooth images where high-frequency details get lost. More expressive models lead to substantially better results (e.g. a hierarchy of latent variables).

A different insight on this, from [link]: Vanilla VAEs with Gaussian posteriors / priors and factorized pixel distributions aren't blurry, they're noisy. People tend to show the mean value of $p(x \mid z)$ rather than drawing samples from it. Hence the reported blurry samples aren't actually samples from the model, and they don't reveal the extent to which variability is captured by pixel noise. Real samples would typically demonstrate salt and pepper noise due to independent samples from the pixel distributions. VAEs are poor models of data whenever insufficiently flexible posterior / prior / decoder distributions are used. These issues are much improved when more expressive choices are used as in IAF, pixelvae, variational lossy autoencoder.

ELBO is a loose bound rather than a tight one.

Kullback-Leibler divergence only calculates similarity, rather than really the Gaussianness of the distribution.

Per-sample loss calculation is a problem; you hope that by drawing enough samples, you get a Gaussian.

5.2 Generative Adversarial Networks

GANs implicitly minimize Jensen-Shannon divergence. JS-divergence is symmetric, smooth and bounded between zero and one.

General idea is to have two networks play a two-player game between generator ($G : \mathbb{R}^D \to X$) and discriminator ($D : X \to [0,1]$). Our goal is to train D and G jointly.

5.2.1 Deriving the GAN objective

Considering only the discriminator;

$$\begin{aligned}\mathcal{L}(D) &= -\frac{1}{2}\left(\mathbb{E}_{x\sim p_d}[\log(D(X))] + \mathbb{E}_{z\sim p_z}[\log(1-D(G(z)))]\right) \\ &= -\frac{1}{2}\left(\mathbb{E}_{x\sim p_d}[\log(D(X))] + \mathbb{E}_{x\sim p_g}[\log(1-D(X))]\right),\end{aligned}$$

using an explicit change of variables as we replace $G(z)$ with X. This assumption will never hold in practice. We then define a value function as

$$V(G,D) = \mathbb{E}_{x\sim p_d}[\log(D(X))] + \mathbb{E}_{x\sim p_g}[\log(1-D(X))]$$

Good G maximizes the loss of D, $G^* = \arg\min_G \max_D V(G,D)$ will fool *any* D

5.2.2 Analysing the optimal value for objective

Let $D_G^* = \arg\max_D V(G,D)$. Then our objective becomes $G^* = \arg\min_G V(G, D_G^*)$. Proposition is that optimum of $V(G,D)$ is $D_G^* = \frac{p_d}{p_d+p_g} \iff p_d = p_g$ We solve the value function

$$V(G,D) = \int_x p_d(x)\log(D(x)) + p_g(x)\log(1-D(x))dx$$

$$\begin{aligned}&f(y) = a\log y + b\log(1-y) \\ &f'(y) = 0 \implies y = \frac{a}{a+b} \\ &f''(y=\frac{a}{a+b}) < 0 \text{ with } a,b \in (0,1) \implies \text{maximum}\end{aligned}$$

D_G^* is unique but can't be calculated in practice.

5.2.3 Finding the best G

$$\begin{aligned}V(D_G^*, G) &= \mathbb{E}_{x\sim p_d}\left[\log\frac{p_d}{p_d+p_g}\right] + \mathbb{E}_{x\sim p_g}\left[\log\left(1-\frac{p_d}{p_d+p_g}\right)\right] \\ &= \mathbb{E}_{x\sim p_d}\left[\log\frac{p_d}{p_d+p_g}\right] + \mathbb{E}_{x\sim p_g}\left[\log\frac{p_g}{p_d+p_g}\right] \\ &= -\log 2 + \mathbb{E}_{x\sim p_d}\left[\log\frac{2p_d}{p_d+p_g}\right] - \log 2 + \mathbb{E}_{x\sim p_g}\left[\log\frac{2p_g}{p_d+p_g}\right] \\ &= -\log 4 + D_{KL}(p_d \parallel \frac{p_d+p_g}{2}) + D_{KL}(p_g \parallel \frac{p_d+p_g}{2}) \\ &= -\log 4 + 2D_{JS}(p_d \parallel p_g)\end{aligned}$$

- Objective minimizes the JS-divergence, with global minimum is achieved at $p_d = p_g$ with $V(\cdot,\cdot) = -\log 4$
- $2D_{JS}(p_d \parallel p_g) \geq 0$ holds, it is $0 \iff p_d = p_g$, which is the Nash equilibrium condition where the discriminator is maximally confused and outputs $1/2$ everywhere

5.2.4 Training GANs

Instead of minimizing D being right, rather maximize D being wrong to get nice steep gradients for G early on, as it is a problem if D is too powerful w.r.t. G in the beginning. Train by alternate between two steps:

1. Gradient ascent on D:

$$\max_{\theta_d} \mathbb{E}_{x\sim p_d(x)}[\log D_{\theta_d}(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D_{\theta_d}(G_{\theta_g}(z)))]$$

2. Gradient ascent on G:

$$\max_{\theta_g} \mathbb{E}_{z\sim p_z(z)}[\log(D_{\theta_d}(G_{\theta_g}(z)))]$$

Issues in training:

- Need to find Nash-Equilibrium in two-player game. Making downhill progress for one player may move the other player uphill.
- Generator can trick discriminator due to underlying image statistics.
- Mode collapse (no sample diversity): saddle point in dual energy landscape.
- JSD correlates badly with sample quality, so we do not know when to stop training. JSD saturates when D tends to be optimal, leading to bad gradients. One solution would be to use a different divergence (e.g. Wasserstein distance, f-divergence).

Comparison with VAEs:

- No variational bound is needed.
- Sharper images, higher image quality.
- GANs only care about samples.
- GANs have [theoretical] ability to model distributions fully; specific model families usable within GANs are already known to be universal approximators, so GANs are already known to be asymptotically consistent. Some VAEs are conjectured to be asymptotically consistent, but this is not yet proven

Example models:

- DC-GAN: fully convolutional D and G
- Pix2pix: conditional adversarial network, requires paired images
- CycleGAN: unpaired image translation through inverse mapping and cycle consistency loss

5.3 Autoregressive models

A regression model, such as linear regression, models an output value based on a linear combination of input values. This technique can be used on time series where input variables are taken as observations at previous time steps, called lag variables. For example, we can predict the value for the next time step $(t + 1)$ given the observations at the last two time

steps ($t-1$ and $t-2$). Because the regression model uses data from the same input variable at previous time steps, it is referred to as an **autoregression** (regression of self). Sequence models are generative models.

Autoregressive property: No computational path between output unit x_d and any of the input units $x_d, \ldots, x_D$ must exist (relative to some ordering). In other words, each output depends only on previous inputs w.r.t some ordering.

Tabular approach over n dimensions correspond to $2^n - 1$ parameters. Fully visible belief networks model via logistic regression: $O(n^2)$.

5.3.1 NADE and MADE

Neural Autoregressive Density Estimator (NADE) is an autoencoder like neural network to learn conditional probability. Each conditional is modeled by the same neural network. NADE is for binary data, but it is easily extendable to other types of observations (e.g. reals, multinomials). Computations are in $O(TD)$ (seqlen times number of samples). Paper explores different orderings of inputs but random order most of the time just works fine.

$$\mathbf{h}^{(k)} = \text{sigm}(\mathbf{b} + \mathbf{W}_{.,<k}\mathbf{x}_{<k})$$
$$\hat{x}_k = \text{sigm}(c_k + \mathbf{V}_{k,.}\mathbf{h}^{(k)})$$

We can leverage the fact that

$$(\mathbf{b} + \mathbf{W}_{.,<k+1}\mathbf{x}_{<k+1}) - (\mathbf{b} + \mathbf{W}_{.,<k}\mathbf{x}_{<k}) = \mathbf{W}_{.,k+1}x_{k+1}$$

The distribution modeled by NADEs has the great advantage to be tractable, since all of its conditional probability distributions are themselves tractable. This means contrary to an RBM, performance can be directly measured via the negative log-likelihood (NLL) of the dataset.

NADE extensions: real-valued NADE (RNADE), order-less and deep NADE (DeepNADE), convolutional NADE (ConvNADE), document NADE (DocNADE) ...

Teacher forcing: Ground truth values of the pixels are used for conditioning when predicting subsequent values. It is done for stabilizing training so that the predictions do not diverge. Inference always uses own predictions (fully generative model).

Masked Autoencoder Distribution Estimator (MADE) constraints an autoencoder to fulfil autoregressive property such that its output can be used as conditional.

For timesteps $t = 1, \ldots, n$ sample from $(1, n-1)$ for each hidden unit. If its sampled value is smaller than the preceding one, drop (mask) the connection.

- Training has same complexity as regular autoencoders, criterion is NLL for binary x
- Computing $p(x)$ is just a matter of performing a forward pass, but sampling requires D forward passes
- In practice, very large hidden layers necessary as not all hidden units can contribute to each conditional

5.3.2 PixelRNN, PixelCNN, Gated PixelCNN

See [link] for detailed animations on the topic.

PixelRNN generates image pixels starting from corner, dependency on previous pixels modeled using an RNN (LSTM). Training maximizes likelihood of training images. Here, sequential generation is slow due to explicit pixel dependencies. PixelRNN uses masked convolutions as a convenient way of enforcing dependencies between pixels. Higher layers are fully connected. It is also autoregressive over color channels; authors actually allow information from R (red) channel go into G (green), and from R and G go into B. So we have ordering not only within spatial dimensions but also within source channels (think colors) too. Different architectures could be used: Row LSTM, Diagonal LSTM, Diagonal BiLSTM ...

PixelCNN again generates image pixels starting from corner, but dependency on previous pixels now modeled using a CNN over context region. PixelCNN lowers the training time considerably as compared to PixelRNN. However, the image generation is still sequential as each pixel needs to given back as input to the network to compute next pixel.

In Vanilla PixelCNN stacked masked convolutions creates blind spots, since convolutional networks capture a bounded receptive field. **Gated PixelCNN** uses a more expressive nonlinearity and two layers of stacks to match the performance of PixelRNN. Information flow between horizontal and vertical stacks preserves the correct pixel dependencies.

- Horizontal Stack: It conditions on the current row and takes as input the output of previous layer as well as the of the vertical stack.
- Vertical Stack: It conditions on all the rows above the current pixel. It doesn't have any masking. It's output is fed into the horizontal stack and the receptive field grows in rectangular fashion.

5.3.3 TCN, VRNN, C-VRNN, STCN

Temporal Causal Networks (TCN):

- Idea is to adapt PixelCNN to work with audio data.
- Problem: much higher dimensionality than images
- Long term temporal dependencies
- Uses dilated convolutions to reach larger receptive fields
- Inference speed
- Note: strided convolutions cannot be used due to the need to preserve resolution

Variational RNNs (VRNN): increase expressive power of RNNs by incorporating stochastic latent variables into hidden state of an RNN, via including one VAE per timestep. Including a dynamic prior (similar to HMMs, Kalman Filters) explicitly models temporal dependencies between timesteps in latent space.

Conditional Variational RNNs (C-VRNN): [For DeepWriting] introduce a new set of latent variables for the model to decouple style and content.

Stochastic Temporal Convolutional Networks (STCN): Combine the computational advantages of TCNs with the expressivity of stochastic RNNs, via integrating a hierarchical VAE with a TCN. It introduces a hierarchy of stochastic latent variables.

5.3.4 Summary of AR generative models

- The basic difference between Generative Adversarial Networks (GANs) and Autoregressive models is that GANs learn implicit data distribution whereas the latter learns an explicit distribution governed by a prior imposed by model structure.
- Nice property of autoregressive models is that because we train via NLL, we also have a direct metric to compare performance. This also makes it straightforward to apply in domains such as compression and probabilistic planning and exploration.
- The training is more stable than GANs (e.g. in PixelRNN or PixelCNN).
- AR models work for both discrete and continuous data, whereas it's hard to learn to generate discrete data for GAN, like text.
- Major downside of AR models is their slow sequential generation.

\+ $p(x)$ is tractable, so easy to train, easy to sample (though slower)

− No natural latent variable representation (but doable, cf. C-VRNN, STCN)

6 Reinforcement Learning

Concepts: agent, environment, state, action, cumulative reward

(Temporal) credit assignment problem: given a long sequence of actions, e.g. sequence of moves in chess that lead to a win or loss, identify which action or actions were useful or useless in obtaining the final feedback. It is problematic for RL scenarios as rewards, especially in fine grained state-action spaces, can occur terribly temporally delayed.

Discount factor tells how important future rewards are to the current state. In other words, it represents how much future events lose their value according to how far away in time they are. The undiscounted formulation is appropriate for episodic tasks, in which the agent–environment interaction breaks naturally into episodes; the discounted formulation is appropriate for continuing tasks, in which the interaction does not naturally break into episodes but continues without limit.

Value function $V_\pi : S \to \mathbb{R}$ of a policy π expresses the expected cumulative reward for each state we achieve when following the policy π.

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a) \left[r + \gamma \, \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s'] \right] \\
&= \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a)[r + \gamma v_\pi(s')], \text{ for all } s \in \mathcal{S}
\end{aligned}
$$

- Discounted return: $G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$
- Value function: $v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s]$
- Action-value function: $q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$
- Greedy policy: $\pi'(s) \doteq \arg\max_a q_\pi(s, a) = \arg\max_a [r(s, a) + \gamma v_\pi(p(s, a))]$

6.1 Bootcamp

The difference between on-policy and off-policy methods is whether you use the same policy to generate state-action pairs during training or you have two different ones, respectively.

6.1.1 Dynamic Programming

Dynamic programming: 3 key ideas for solving MDPs

- Iterative Policy Evaluation
  (a) Compute the state value function v_π for any arbitrary policy π
- Policy Iteration for estimating $\pi \approx \pi^*$
  (a) Compute the state value function v_π for any arbitrary policy π (iterate value function update or solve linear system of equations)
  (b) Update policy π given $v_\pi \to \pi'$
  (c) Iterate till $\pi \approx \pi'$
- Value Iteration for estimating $\pi \approx \pi^*$
  (a) Compute optimal state value function v^*
  (b) Compute policy based on v^*

Pros and Cons of Dynamic Programming

+ Exact methods
+ Policy/value iteration are guaranteed to converge in finite number of iterations
+ Easy to implement
− Need to know the transition probability matrix
− Need to iterate over the whole state space (very expensive)
− Requires memory proportional to the size of the state space

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set.

Value iteration is typically more efficient

6.1.2 Monte Carlo Sampling

Monte Carlo Methods: If the state space is too big, we can never visit all the states. Even over many episodes we will typically only visit a small subset of the state space. The

value function is defined as an expectation, so we can just run episodes with the given policy and compute samples of the expression in the expectation.

Pros and Cons of Monte Carlo Sampling

+ Unbiased estimate
+ Do not need to know system dynamics
− High variance
− Exploration/Exploitation dilemma
− Need termination state, slow for long episodes (MC must wait with the update until the final outcome!)

The first-visit MC method estimates $v_\pi(s)$ as the average of the returns following first visits to s, whereas the every-visit MC method averages the returns following all visits to s. Both first-visit MC and every-visit MC converge to $v_\pi(s)$ as the number of visits (or first visits) to s goes to infinity.

Another advantage of MC is that it is easy and efficient to focus Monte Carlo methods on a small subset of the states. A region of special interest can be accurately evaluated without going to the expense of accurately evaluating the rest of the state set.

6.1.3 Temporal Difference Learning

Temporal Difference Learning: Instead of computing our value estimates from whole sequences, we can also try to estimate it based on incremental rewards. By increasing the number of episodes the estimates get more refined. TD methods are model-free and guaranteed to converge. Unlike MC methods, they can learn before knowing the final outcome (meaning less memory and less peak computation) and even in the absence of it (i.e. from incomplete sequences).

In **TD(0) learning** (one-step TD) we take a step and update the value function. While Monte Carlo methods only adjust their estimates once the final outcome is known, TD methods adjust predictions to match later, more accurate, predictions about the future before the final outcome is known.

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

SARSA computes the Q-Value according to a policy and then the agent follows that policy. A SARSA (State-action-reward-state-action) agent interacts with the environment and updates the policy based on actions taken, hence this is known as an on-policy learning algorithm.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

As in all on-policy methods, we continually estimate q_π for the behaviour policy π, and at the same time change π toward greediness with respect to q_π. The convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on Q. For example, one could use ε-greedy or ε-soft policies.

Q-learning is off-policy, it computes the q-value according to a greedy policy, but the agent follows a different exploration policy. During training (i.e. collecting data) Q-learning uses an ϵ-greedy policy to explore state-space.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

In Q-learning, unlike SARSA where an action A' was chosen by following a certain policy, the action A' (a in this case) is chosen in a greedy fashion by simply taking the max of Q over it.

TD-Lambda: the lambda (λ) parameter refers to the trace decay parameter, with $0 \leq \lambda \leq 1$. Higher settings lead to longer lasting traces; that is, a larger proportion of credit from a reward can be given to more distant states and actions when λ is higher, with $\lambda = 1$ producing parallel learning to Monte Carlo RL algorithms.

Pros and Cons of Temporal Difference Learning

+ Less variance than Monte Carlo Sampling due to bootstrapping
+ More sample efficient
+ Do not need to know the transition probability matrix
− Biased due to bootstrapping
− Exploration/Exploitation dilemma
− Can behave poorly in stochastic environments

Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).

6.1.4 Summary

- Dynamic Programming
  - Uses value of neighbouring states to update current state (given π)
  - Visits every state
  - Requires knowledge of the transition matrix
  - Bootstraps, does not sample
- Monte Carlo Methods
  - Run in episodes, update state values at end of episode.
  - Experience based (no need to know system dynamics)
  - Needs termination and can be slow for long episodes
  - High Variance
  - Does not bootstrap, samples
- Temporal Differences
  - Run in episodes, update state values at n-step.

- Combination of DP and MC ideas
- Exploration/Exploitation → Local minima
- Bootstraps and samples

6.2 Deep Reinforcement Learning

Directly optimising the policy (e.g. finding the best policy from a collection of policies) could be easier than exploring the full state-action space, as in Q-learning. Exploration of full state-action space is a high-dimensional problem and intractable for continuous action space.

All methods that follow this general schema we call policy gradient methods, whether or not they also learn an approximate value function. Methods that learn approximations to both policy and value functions are often called actor–critic methods, where 'actor' is a reference to the learned policy, and 'critic' refers to the learned value function, usually a state-value function.

Policy gradients tries to update the policy so that good trajectories are made more likely and vice versa. For a trajectory τ, it pushes up the probabilities of the actions seen if $r(\tau)$ is high and pushes them down if $r(\tau)$ is low. Policy is a stochastic function approximator parametrized via a neural network and policy update is just gradient ascent.

Off-policy methods typically leads to better exploration but they are limited by the types of problems to apply, because they are data intensive.

Comparison of policy optimization and dynamic programming:

- Policy optimization
  - Directly optimizes the desired quantity
  - More compatible with modern ML machinery, including NN and recurrence
  - More versatile and flexible
  - More compatible with auxiliary objectives
- Dynamic programming
  - Indirect, exploits problem structure and self-consistency
  - More compatible with off-policy and exploration
  - More sample efficient (when they work)

REINFORCE is the backbone of modern RL algorithms. It is unbiased but has high variance. Updates on the gradients of the performance measure can be done without knowledge of the underlying environment dynamics, i.e. REINFORCE trick makes the updates independent from the transition probabilities.

$$\text{Expected reward is } J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)}[r(\tau)] = \int_{\tau} r(\tau) p(\tau;\theta) d\tau$$

Its differentiation is intractable as gradients of an expectation is problematic when p depends

on θ. This is where REINFORCE trick comes into play:

$$\nabla_\theta J(\theta) = \int_\tau r(\tau)\nabla_\theta p(\tau;\theta)d\tau = \int_\tau r(\tau)p(\tau;\theta)\frac{\nabla_\theta p(\tau;\theta)}{p(\tau;\theta)}d\tau = \int_\tau r(\tau)p(\tau;\theta)\nabla_\theta \log p(\tau;\theta)d\tau$$

We can estimate $\nabla_\theta J(\theta) = \int_\tau r(\tau)p(\tau;\theta)\nabla_\theta \log p(\tau;\theta)d\tau = \mathbb{E}_{\tau\sim p(\tau;\theta)}[r(\tau)\nabla_\theta \log p(\tau;\theta)]$ with Monte Carlo sampling. Independence from transition probabilities can be seen as below

$$\begin{aligned} p(\tau;\theta) &= \prod_{t\geq 0} p(s_{t+1}|s_t,a_t)\pi_\theta(a_t|s_t) \\ \log p(\tau;\theta) &= \sum_{t\geq 0} \log p(s_{t+1}|s_t,a_t) + \log \pi_\theta(a_t|s_t) \\ \nabla_\theta \log p(\tau;\theta) &= \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(a_t|s_t) \end{aligned}$$

We introduce a baseline for the variance reduction to ease credit assignment problem as the raw value of a trajectory is not necessarily meaningful (e.g. all cumulative rewards could be positive or non-positive). A simple baseline would be the constant moving average of rewards experienced so far from all trajectories.

Actor-Critic Algorithm combines ideas from policy gradients and Q-learning by training both an actor (the policy) and a critic (the Q-function). The actor decides which action to take, and the critic tells the actor how good its action was and how it should adjust. Defining an advantage function (cf. TD-error) $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$ the estimator becomes

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0} A^{\pi_\theta}(s_t,a_t)\nabla_\theta \log \pi_\theta(a_t|s_t)$$

Idea is to use expected value of what we should get from that state as the baseline in policy gradients. Task of the critic is alleviated the as it only has to learn the values of (state, action) pairs generated by the policy. We can also incorporate Q-learning tricks, e.g. experience replay. Extensions of Actor-Critic Algorithm

- Trust Region Policy Optimization (TRPO): Poses policy minimization as constrained optimization problem which includes a KL-divergence of old and new policies. The idea is to control step length by preventing the action distribution to change too much.
- Proximal Policy Optimizations (PPO): Follow up work of TRPO, but using first order optimization (less computationally heavy). Relaxes the hard constraint condition and turns it into a tunable parameter. Trade off between fast optimization and allowing some bad updates. However, we can even do better by using a clipped objective, rather than using KL-divergence.
- Deep Deterministic Policy Gradients (DDPG): DDPG is an actor critic off policy DRL algorithm that combines (Deep) Q-learning and Deterministic Policy Gradients (DPG). It adds random noise when selecting an action for data collection.

Issues in DRL research

- Exploration vs. Exploitation: Trade off between when to explore and when to exploit, Agents can easily get stuck in local minima (i.e. exploit behaviour that is irreversible)

- Reward Hacking: Rewards have to be carefully designed , undesired behaviour may occur because of ill shaped reward
- Sample Efficiency: DRL often needs an excessive amount of samples to learn a policy. Especially on policy methods suffer from sample inefficiency, because the samples have to be "thrown away" after the policy is updated
- Domain Knowledge: Even small implementation differences can have a big impact

Sample efficiency issue: The problem in RL is that it requires an enormous amount of data to successfully train policies. The reason being that as we update and improve our policies, we can also collect better data. Therefore, old data has to be thrown away and replaced by improved data samples, which makes it sample inefficient (especially on-policy methods suffer from this, as collected data is essentially only used once and then thrown away).

7 Appendix

Kullback-Leibler divergence:

$$D_{KL}(p \parallel q) = \int_x \log \frac{p(x)}{q(x)} dx$$

Jensen-Shannon divergence:

$$D_{JS}(p \parallel q) = \frac{1}{2} D_{KL}(p \parallel \frac{p+q}{2}) + \frac{1}{2} D_{KL}(q \parallel \frac{p+q}{2})$$

7.1 TensorFlow tips

When you build a model there are usually some design choices and hyper-parameters that you want to experiment with. Hence, it is good practice to make those parameters configurable through the command line or an external configuration file. TensorFlow provides built-in support for this, called **FLAGS**.

sparse_softmax_cross_entropy_with_logits: The problem with the cross-entropy, as mentioned before, is that it is numerically unstable and can produce inf values during training. Hence, TensorFlow produces a **more stable** version which takes as input the logits, not the softmax activated values. This is why we did not choose an activation function for the outputs of the dense layer.

In TensorFlow, a tensor has both a static (inferred) shape and a dynamic (true) shape. The **static** shape can be read using the tf.Tensor.**get_shape**() method: this shape is inferred from the operations that were used to create the tensor, and may be partially complete. If the static shape is not fully defined, the **dynamic** shape of a Tensor t can be determined by evaluating tf.**shape**(t).

- We can use **in_top_k** function to get the accuracy.
- **global_step** variable is just there to act as a counter to SGD.
- VGG generalizes really well, keras even lets you use the ImageNet weights (also ResNet), it is a good starting point for the projects.

7.2 Reparametrization trick

Given Gaussian with μ and σ^2 as $z \sim \mathcal{N}(\mu, \sigma^2)$. Value of z changes every time we sample it. Reparametrization trick assumes $\exists$ underlying random variable $\epsilon \sim \mathcal{N}(0, 1)$, $z = \mu + \sigma\epsilon$. ϵ is a random variable but not a function of μ or σ. We can now take derivative of $z = f(x, \epsilon, \theta)$ w.r.t. μ, σ. Tells us how an infinitesimal change in μ and σ affects the output if we keep ϵ fixed.

We use the reparameterization trick to express a gradient of an expectation as an expectation of a gradient. The issue is not that we cannot backprop through a "random node" in any technical sense. Rather, backproping would not compute an estimate of the derivative. Without the reparameterization trick, we have no guarantee that sampling large numbers of z will help converge to the right estimate of $\nabla\theta$. We use the reparameterization trick to express a gradient of an expectation as an expectation of a gradient. [link]