Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Semi-supervised learning with Deep Generative Models

Prueba con imágenes hiperespectrales

Autor: Fabián Souto H.
Profesor guía: Pablo Guerrero
Santiago, Chile

Resumen

En el marco de clasificación de materiales con imágenes hiperespectrales, es de alto interés ser capaces de poder clasificar automáticamente, es decir, poder ocupar alguna herramienta de Machine Learning. En vista de que se cuenta con una gran cantidad de datos pero muy pocos etiquetados, resulta útil tener algún mecanismo que aprenda tanto de los no etiquetados como de los que sí se encuentran con su *label*.

Aquí es donde aparecen los *Variational Autoencoders* (VAE) y los *Deep Generative Models.* Gracias a D.P. Kingma, D.J. Rezende, S-Mohamed y M. Welling, quienes desarrollaron estas ideas, es que se puede hacer un entrenamiento "semi supervisado".

La primera etapa consta de una extracción de características con un VAE y la segunda etapa es hacer clasificación con algún clasificador conocido o también con una red neuronal.

A lo largo de este informe desarrollaremos las principales ideas detras de estos modelos, veremos una implementación de ellos, cómo poder ocuparlos para nuestro problema y los resultados obtenidos.

Índice de Contenidos

Lista de Figuras

Lista de Tablas

1. Deep Generative Models

Nos enfrentamos a un problema donde tenemos que los datos están etiquetados. Tenemos un vector de características $x_i \in R^D$ junto a su etiqueta y_i. Diremos que las observaciones (los x_i) tienen variables latentes que notaremos por z_i. Como nos enfrentamos al problema semi-supervisado quiere decir que no todos nuetros datos están etiquetados. Diremos que sobre los datos vamos a tener una distribución empírica

$$p_l(x, y)$$

para los datos etiquetados (l: labeled) y

$$p_u(x)$$

para los no etiquetados (u: unlabeled). Ahora veremos modelos para el aprendizaje semi-supervisado que explotan descripciones generativas de los datos para mejorar la clasificación que se será obtenida sólo de los que sí tienen etiquetas.

1.1. Latent-feature discriminative model (M1)

Es típico ocupar un modelo que obtiene una representación característica de los datos. Usando estas características se entrena un clasificador. Esta extracción de características permite poder hacer un clustering de observaciones relativas en un *latent feature space* que permite clasificación más precisa, incluso con un número pequeño de etiquetas. En vez de ocupar un auto-encoder prefieren ocupar un modelo generativo profundo que les permite extraer de mejor manera estas *latent features*.

El modelo consiste en

$$p(z) = N(z|0, I)$$

$$p_\theta(x|z) = f(x; z, \theta)$$

donde f es en realidad cualquier *likelihood* que queramos ocupar. Sus probabilidades se forman por una transformación no-lineal, con parámetros θ, de un conjunto de variables latentes z. Esta tranformación no-lineal es la que permite obtener *higher moments* de los datos (me imagino que mayores niveles de abstracción de las características) y escogieron que estas transformaciones no lineales sean con *deep neural networks*.

Ahora los samples aproximados del *posterior* $p(z|x)$ se usan como características para entrenar un clasificador para predecir la clase y. Haciendo esto entonces pueden hacer la clasificación en un espacio con menos dimensiones, porque se espera ocupar menos variables latentes que número de características tiene la obvservación. Como estamos en un espacio con dimensiones menores y más encima nuestro *posterior* son gaussianas independientes cuyos parámetros se forman por las transformaciones no-lineales se obtiene que los ejemplos son más fáciles de separar. Este sencillo paso resulta en mejoras en la clasificación de las SVMs.

1.2. Generative semi-supervised model (M2)

Se propone un modelo probabilístico que describe los datos como generados por una clase latente y en adición a una variable latente z. Los datos son explicados por el modelo generativo

$$p(y) = Cat(y|\pi)$$

$$p(z) = N(z|0, I)$$

$$p_\theta(x|y, z) = f(x; y, z, \theta)$$

donde $Cat(y|\pi)$ es la distribución *categorical*. La variable de las clases y se trata como latente si no está disponible y z son variables latentes adicionales. Estas variables latentes son marginalmente independientes, entonces permiten separar, en casos de lectura de dígitos por ejemplo, el dígito que se está leyendo (la clase y) del estilo de escritura (variable z). Como en el caso anterior el likelihood es seteado por una transformación no-lineal de las variables latentes. La transformación no lineal la hacen con deep neural netowrks. La clasificación entonces en este modelo se hace a través de inferencia. Para saber los labels que faltan se calculan los posterior $p_\theta(y|x)$.

1.3. Stacked generative semi-supervised model (M1+M2)

Combinación de ambos modelos. Primero se obtienen variables latentes z_1 con M1 y entrenar el modelo M2 con esos *features* en vez de los datos brutos. El resultado es un modelo generativo con dos capas de variables estocásticas:

$$p_\theta(x, y, z1, z2) = p(y)p(z_2)p_\theta(z_1|y, z_2)p_\theta(x|z_1)$$

donde los *priors* $p(y)$ y $p(z_2)$ son igual que en el modelo anterior, las otras dos son parametrizadas con *deep neural networks.*

2. Scalable variational inference

2.1. Lower bound objective

En todos estos modelos la computación exacta de la distribución a posterior es intratable debido a la no-linealidad y las dependencias entre las variables aleatorias. Para los modelos descritos introduciremos una distribución $q_\phi(z|x)$ con parámetros ϕ que aproximan la distribución a posterior $p(z|x)$. Luego, siguiendo el principio variacional obtendremos una cota inferior del *marginal likelihood*, lo que asegurará que la aproximación se parezca a la distribución real.

Se construye la distribución a posterior aproximada $q_\phi(\cdot)$ un *inference or recognition model* que se ha transformado en un método popular para inferencia variacional eficiente. Usando una *inference network* nos saltamos la necesidad de calcular para cada punto los parámetros variacionales pero podemos obtener de igual forma los parámetros ϕ que son globales. Esto nos permite disminuir los

costos de la inferencia generalizando la distribución a posterior estimada para todas las variables latentes a través de los parámetros de la red, lo que permite inferencia rápida tanto en entrenamiento como en testing.

Una *inference netowrk* es introducida para todas las variables latentes y son parametrizadas con *deep neural networks* cuyos outputs forman los parámetros de la distribución $q_\phi(\cdot)$.

Para el *latent-feature discriminative model* (M1) se usa una *Gaussian inference network* $q_\phi(z|x)$ para las variables latentes z.

Para el *generative semi-supervised model* (M2) se introduce un *inference model* para cada una de las variables latentes, z e y, que se asumen tienen una forma factorizable $q_\phi(z, y|x) = q_\phi(z|x)q_\phi(y|x)$, que son distribuciones gaussianas y categorical respectivamente.

M1:

$$q_\phi(z|x) = \mathcal{N}(z|\mu_\phi(x), diag(\sigma^2_\phi(x)))$$

M2:

$$q_\phi(z|y, x) = \mathcal{N}(z|\mu_\phi(y, x), diag(\sigma^2_\phi(x)))$$

$$q_\phi(y|x) = \text{Cat}(y|\pi_\phi(x))$$

donde $\sigma_\phi(x)$ es un vector de desviaciones estándar, $\pi_\phi(x)$ es un vector de probabilidades, y las funciones $\mu_\phi(x)$, $\sigma_\phi(x)$ y $\pi_\phi(x)$ son representadas como MLP.

2.2. Latent feature discriminative model objective

El *variational bound* $\mathcal{J}(x)$ sobre el marginal likelihood para un solo punto es

$$\log p_\theta(x) \geq \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - \text{KL}[q_\phi(z|x)||p_\theta(z)] = -\mathcal{J}(x)$$

La *inference network* $q_\phi(z|x)$ es usada durante el entrenamiento del modelo usando tanto los ejemplos etiquetados como los no etiquetados. Esta distribución a posterior aproximada es usada entonces como extractor de características de los ejemplos etiquetados y esas características entonces son usadas para entrenar un clasificador.

2.3. Generative semi-supervised model objective

Para este modelo tenemos que considerar dos casos. El primer caso, el label correspondiente del dato se conoce y el *variational bound* es una simple extensión del bound puesto más arriba (el del modelo anterior):

$$\log p_\theta(x, y) \geq \mathbb{E}_{q_\phi(z|x,y)}[\log p_\theta(x|y, z) + \log p_\theta(y) + \log p(z) - \log q_\phi(z|x, y)] = -\mathcal{L}(x, y)$$

Para el caso en que la etiqueta no la conozcamos, la clase es tratada como una variable latente donde se hace inferencia a posterior y la cota entonces es:

$$\log p_\theta(x) \geq \mathbb{E}_{q_\phi(y,z|x)}[\log p_\theta(x|y,z) + \log p_\theta(y) + \log p(z) - \log q_\phi(y,z|x)]$$

$$= \sum_y q_\phi(y|x)(-\mathcal{L}(x,y)) + \mathcal{H}(q_\phi(y|x)) = -\mathcal{U}(x)$$

Atención acá, que la esperanza si sabemos la etiqueta se calcula sobre $q_\phi(z|x,y)$ y si no sabemos la etiqueta se calcula sobre $q_\phi(y,z|x)$, pero habíamos dicho que

$$q_\phi(z,y|x) = q_\phi(z|x)q_\phi(y|x)$$

Entonces si notamos bien la esperanza se puede interpretar como que es el término $-\mathcal{L}(x,y)$ pero falta multiplicarlo por $q_\phi(y|x)$ y marginalizar sobre todas las clases. Además sobra un término: $-\log q_\phi(y|x)$ (al factorizar el $-\log q_\phi(y,z|x) = -\log q_\phi(y|x) - \log q_\phi(z|x))$ que al ser multiplicado por el $q_\phi(y|x)$ resulta la entropía, entonces esta es sumada para mantener la igualdad. Es por esto que la esperanza que se muestra es igual a lo que se pone justo abajo.

Por lo tanto la cota para el marginal likelihood para todo el data set será

$$\mathcal{J} = \sum_{(x,y)-p_l} \mathcal{L}(x,y) + \sum_{x-p_u} \mathcal{U}(x)$$

La distribución $q_\phi(y|x)$ para los ejemplos que no tienen etiqueta es tratada como un *discriminative classifier* y podemos usar este conocimiento para construir el mejor clasificador posible como nuestro *inference model*. Esta distribución es la que se ocupa en los test para predicciones en los datos que no se han visto.

En la función objetivo $\mathcal{J}$ la distribución predictiva de las etiquetas $q_\phi(y|x)$ contribuye sólo al segundo término, al de los que no tienen etiquetas, lo que es una propiedad no deseable si queremos usar esta distribución como nuestro clasificador. Idealmente, todo los parámetros (tanto del modelo como variacionales) deberían ser aprendidos en todos los casos. Para arreglar esto se agrega una *classification loss* cuya distribución $q_\phi(y|x)$ también aprende de los ejemplos etiquetados.

La función objetivo final resulta ser:

$$\mathcal{J}^\alpha = \mathcal{J} + \alpha \cdot \mathbb{E}_{p_l(x,y)}[-\log q_\phi(y|x)]$$

Notemos que la esperanza se calcula sobre la distribución de los ejemplos etiquetados.

Acá el hiper-parámetro α controla el peso relativo entre *generative or purely discriminative learning*. En los experimentos usan un $\alpha = 0{,}1 \cdot N$.

3. Optimization

Los bounds encontrados para nuestros modelos proveen una función objetivo unificada para la optimización de ambos parámetros, θ y ϕ, del *generative* y del *inference model* respectivamente. Esta optimización se puede hacer en conjunto, sin tener que recurrir al algoritmo EM (expected maximization), una reparametrización determinística de las esperanzas en la función objetivo en conjunto con Monte carlo approximation.

A continuación se describe las estrategias principales para el M1, pues las mismas se ocupan para el M2.

Cuando el prior $p(z)$ es una gaussiana esférica $p(z) = \mathcal{N}(z|0, I)$, y la distribución variacional $q_\phi(z|x)$ es una gaussiana (como la que habíamos mencionado más arriba) el término de la divergencia KL puede ser computado analíticamente.

Ahora, el otro término, el del log-likelihood se puede reescribir como

$$\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] = \mathbb{E}_{\mathcal{N}(\epsilon|0,I)}[\log p_\theta(x|\mu_\phi(x) + \sigma_\phi(x) \cdot \epsilon)]$$

Donde el término $\cdot$ indica la multiplicación término a término. Esa transformación es la que mencionan en el paper de los DLGMs que dicen que cualquier gaussiana puede ser obtenida desde una esférica con esa transformación.

Esta esperanza aún no se puede resolver analíticamente (no lo vamos a hacer) pero sus gradientes respecto a los parámetros ϕ y θ pueden ser computados eficientemente como esperanzas de unos gradientes:

$$\nabla_{\{\phi,\theta\}}\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] = \mathbb{E}_{\mathcal{N}(\epsilon|0,I)}[\nabla_{\{\phi,\theta\}} \log p_\theta(x|\mu_\phi(x) + \sigma_\phi(x) \cdot \epsilon)]$$

Los gradientes de la función de costo para el M2 pueden ser calculados por aplicación directa de la regla de la cadena y notando que el límite condicional $\mathcal{L}(x_n, y)$ contiene los mismos términos.

Durante la optimización se usan estos gradientes estimados en conjunto con métodos de descenso del gradiente estocástico, como SGD, RMSprop, o ADAGrad. Esto resulta en actualización de parámetros del estilo $(\theta^{t+1}, \phi^{t+1}) \leftarrow (\theta^t, \phi^t) + \Gamma^t(g^t_\theta, g^t_\phi)$ donde Γ es una *diagonal preconditioning matrix* que va adaptando el gradiente para minimización más rápida.

4. Código

4.1. Repositorios

Los autores del paper hicieron público su código y fue en este en el cual me basé para poder hacer una adaptación a los datos con los que contamos nosotros. El fork correspondiente, el cual modifqué es `https://github.com/SetaSouto/nips14-ssl`.
Además cuento con un repositorio en donde se encuentra alojado este informe y otros apuntes en formato Markdown (más informales) que contienen ideas mías y anotaciones respecto a cómo ocupar el código. El repo de los apuntes es `https://github.com/SetaSouto/TrabajoDirigido`.

4.2. HyperspectralData class

Para el manejo y transformaciones de los datos se creó la el archivo *hyperspectralData.py* que contiene la clase *HyperspectralData* la cual se debe inicializar con el path correspondiente a donde se encuentran los datos etiquetados, es decir la carpeta *Labeled HSI*. Por defecto tiene la ruta de mi computador.
Los métodos de la clase se explican por sí solos, están todos documentos y comentados.
La idea que tiene que quedar en mente si es que los modelos reciben como entrada matrices con todos los datos pero separados en dos: Datos y Etiquetas. Las etiquetas deben estar en formato *one hot encoding* y cada etiqueta en una columna, es decir la matriz es de la forma $number\ of\ classes \times number\ of\ samples$. Lo mismo sucede con los datos, que son vectores con 268 datos, por lo tanto se debe entregar una matriz de $268 \times\ number\ of\ samples$. Los datos deben estar **normalizados**, es decir ninguno puede ser mayor que 1,0.
También a considerar, si se acude a las líneas 43 a 49 del archivo *gpulearn_z_x_hyp.py* (que es quien hace el entrenamiento del M1, lo veremos más adelante), se nota que se le dice al modelo que los datos tienen la forma $(67, 4)$ lo que sirve solo para poder "dibujar"lo que el VAE está produciendo. Solamente queda decir que los modelos reciben los datos así como se indicó pero se les alimenta con un par de matrices para entrenamiento, otro para testing y otro para validación; por lo tanto, reciben 6 matrices de input los modelos.

4.3. Entrenamiento M1

Para ejecutar el script de entrenamiento se debe llamar de la siguiente forma:

```
THEANO_FLAGS=floatX=float32 python run_VAE_hyp.pyL
```

Que ejecuta el archivo indicado pero que en realidad llama al main del archivo *gpulearn_z_x_hyp.py* donde se encuentra todo el código, es ahí donde se pueden alterar los **hiperparámetros**, cambiar **los datos de entrenamiento** u otra modificación. Muestra en pantalla cada diez pasos algunas estadísticas como:

- El paso en el que va.

- El tiempo que ha pasado.
- El loglikelihood del batch de entrenamiento.
- El logliklihood del valid set.
- Cuantos pasos lleva sin mejorar.

Respecto a lo último el script se detiene cuando hace 100 pasos sin mejoras. Por experiencia propia hay veces que llega como hatsa 70 pasos sin mejorar y mejora sopresivamente.

Finalmente los resultados se van "mostrando.[en] la carpeta que se indica al comienzo con "logdir". Básicamente es como qué archivo se está ejecutando, con cuantos nodos ocultos y el tiempo actual. En esa carpeta se generan imágenes que son los samples de la red, a medida que aprende es capaz de generar mejore samples, o sea samples más parecidos con los que entrena.

El código del script está basado en el *ru_gpulearn_z_x* del repositorio original.

Solamente cabe mencionar que que este entrenamiento no necesita las etiquetas pero igual son provistas.

4.4. Entrenamiento M2

El entrenamiento del M2 es similar al del M1, solo que en este caso se debe llamar al archivo *run_M2_hyp.py* que ejecuta el main de *learn_yz_x_hyp.py*, es ahí donde se deben hacer las modificaciones como lo indicado en la sección anterior.

Importante: Para entrenar este modelo ya debe haber entrenado el M1 y los diccionarios con las variables de las redes (que en el código son las v y las w) deben estar en el directorio *results/hyper_50-(500, 500)_longrun/*. Esto es porque en el entrenamiento del M2 e ocupa la extracción de características que se puede hacer con M1.

4.5. Extracción de características

Para entender como se hace la extracción de características hice un script llamado $extract\_features\_hyp.py$. Lo iré explicando:

- Primero tenemos que setear el path donde se encuentran los resultados del entrenamiento. En esa carpeta aparte de los samples generados por la red se encuentran los parámetros de la red. Las variables v son para la generación de las características (inference), mientras que los parámetros w (generative) son para generar samples a partir de las variables latentes (características). Hay que tener claro algo aquí:
- Las características no son algo así concreto o determinista, más bien, siendo rigurosos, son las variables latentes que el modelo aprende, que supuestamente generan los ejemplos reales. La idea es "descubrir"los valores de las variables latentes (que en este caso son distribuciones normales, entonces aprendemos sus parámetros, la media y la varianza) para tener una

representación más general de los datos. Es decir, aprendemos como distribuyen estas características más generales, y así, sampleando de estas distribuciones, es capaz de generar nuevos datos, he ahí la razón de generar samples al entrenar el M1.

- Con el path, cargamos las variables del modelo que entrenamos.
- Importamos y cargamos el modelo. Ojo aquí: Hay que setearlos con los mismos hyperparámetros que se ocuparon para entrenar. Los que dejé yo son los que me dieron mejores resultados, pero al ojo, viendo los samples que me generaban, con otros hyperparámetros se generaban imágenes más ruidosas.
- Luego importamos los datos a los cuales les queremos sacar características.
- Después, "transformamos los datos", ocupando la función *dist_qz* que nos entrega la media y la varianza para cada uno de los datos pasados. La función la verdad aún no tengo claro porqué se llama con esos argumentos, pero así la ocupan en el paper y su implementación.
- Finalmente, para extraer las características simplemente sampleamos de las distribuciones, o sea generamos la distribución normal con los parámetros que nos entrega el modelo y sampleamos.

5. Resultados

Los entrenamiento son muy lentos, es por que esto que los resultados no son demasiados, pues para obtener conclusiones de un primer entrenamiento se debe dejar el computador iterando más de 15 horas al menos.

5.1. Primer long run M2

La primera vez que se corrió el M2 después de entrenar el M1 fue muy esperanzador. Los resultados indicaban una precisión de más del 90 %. Hasta que generé en extracto de código que me mostraba qué estaba prediciendo y para mi sorpresa, sólo predecía la etiqueta 39.
¿Qué pasó? No había analizado cómo se distribuían las etiquetas en el archivo que estaba ocupando, claro cerca del 90 % eran la etiqueta 39, respuesta fácil: Todos son 39.

5.2. Segundo long run M2

Producto de esta confusión y de lo ingenuo que fui al no revisar cómo se ditribuían las etiquetas cree una nueva forma para alimentar el M2 que lee de varios archivos datos pero si en algún momento tiene más de 5000 datos de una etiqueta entonces descarta ejemplos al azar y se queda con un máximo de 5000. Así no tenemos problemas de que la mayoría de los datos sean de una sola etiqueta.

Esto llevó a tener mejor precisión, pero aún así no se logró superar el 40 % de efectividad.

6. Errores y trabajo a seguir

El modelo se puede ocupar, se entrena y aprende, pero ¿Por qué no funcionó como se esperaba? Tengo un par de hipótesis -que por el tiempo que toma entrenar estos modelos se deja como trabajo a seguir-.

- Pocas horas de entrenamiento. En el segundo long run del M2 se dejó 16 horas, alcanzó a hacer 46 pasos y lo mejor que obtuvo fue en el paso 34 un 60 % de error en el test set y en el valid set.
- Pero la hipótesis más importante es que solo cambié los datos con que se entrenaba el M2, el extractor de características seguía siendo el que se había entrenado sólo sobre la etiqueta 39.

Es por esto que mi recomendación sería alimentar el M1 de la nueva forma, como se alimentó el M2 (es el método $load\_dataset\_m2$ de $HyperspectralData$), dejarlo correr durante al menos 3 días y lo mismo con el M2.