CS-UY 4563: Lecture 17
Neural Networks cont.

NYU Tandon School of Engineering, Prof. Christopher Musco

COURSE LOGISTICS

- Lab `lab_mnist_partial.ipynb` due Thursday, 4/9.
- Project Proposal due next Monday, 4/13.
    - See guidelines for what to include at: `https://www.chrismusco.com/introml/project_guidelines.pdf`
    - Can be crudely formatted. A shared Google doc is fine, or email me a PDF.
    - I'm seeing lots of really cool project ideas!

TRAINING NEURAL NETWORKS

Two main algorithmic tools for training neural network models:

1. Stochastic gradient descent.
2. Back-propogation.

TRAINING NEURAL NETWORKS

Let $f(\vec{\theta}, \vec{x})$ be our neural network.

$\mathbf{W}_i$ and $\vec{b}_i$ are the weight matrix and bias vector for layer i and g_i is the non-linearity (e.g. sigmoid). $\vec{\theta} = [\mathbf{W}_0, \vec{b}_0, \ldots, \mathbf{W}_\ell, \vec{b}_\ell]$ is a vector of all entries in these matrices.

Goal: Given training data $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)$ minimize the loss

$$\mathcal{L}(\vec{\theta}) = \sum_{i=1}^{n} L\left(y_i, f(\vec{\theta}, \vec{x}_i)\right)$$

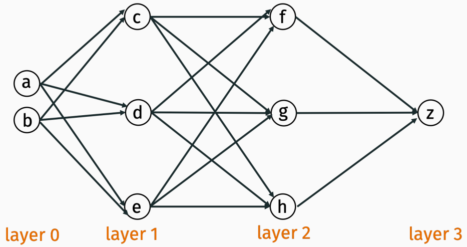
To do so, we need to compute $\nabla L\left(y_i, f(\vec{\theta}, \vec{x}_i)\right)$ for all i.

GRADIENT OF THE LOSS

Last lecture: Reduced our goal to computing $\nabla f(\vec{\theta}, \vec{x})$, where the gradient is with respect to the parameters $\vec{\theta}$.

This will be done using backprop.

BACKPROP EXAMPLE



Notation for next few slides:

- $a, b, \ldots, z$ are the node names, and used to denote values at nodes after applying non-linearity.
- $\bar{a}, \bar{b}, \ldots, \bar{z}$ denote value before applying non-linearity.
- $W_{i,j}$ is the weight of edge from node i to node j.
- $s(\cdot) : \mathbb{R} \rightarrow \mathbb{R}$ is the non-linear activation function.
- β_j is the bias for node j.

Example: $h = s(\bar{h}) = s(c \cdot W_{c,h} + d \cdot W_{d,h} + e \cdot W_{e,h} + \beta_h)$

BACKPROP EXAMPLE

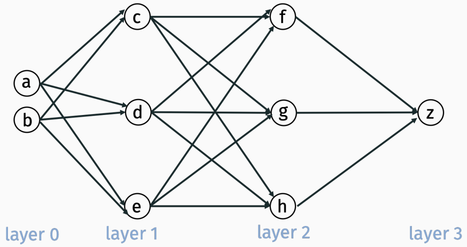
Goal: Compute the gradient $\nabla f(\vec{\theta}, \vec{x})$, which contains the partial derivatives with respect to <u>every</u> parameter:

- $\partial z / \partial \beta_z$
- $\partial z / \partial W_{f,z}$, $\partial z / \partial W_{g,z}$, $\partial z / \partial W_{h,z}$
- $\partial z / \partial W_{c,f}$, $\partial z / \partial W_{c,g}$, $\partial z / \partial W_{c,h}$
- $\partial z / \partial W_{d,f}$, $\partial z / \partial W_{d,g}$, $\partial z / \partial W_{d,h}$
- $\vdots$
- $\partial z / \partial W_{a,c}$, $\partial z / \partial W_{a,d}$, $\partial z / \partial W_{a,e}$

Two steps: <u>Forward pass</u> to compute function value. <u>Backwards pass</u> to compute gradients.

BACKPROP EXAMPLE

Step 1: Forward pass.

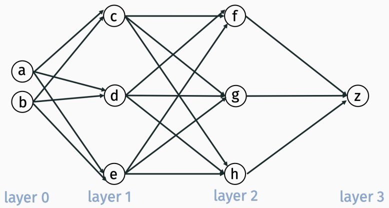


- Using **current parameters**, compute the output z by moving from left to right.
- Store all intermediate results:

$$\bar{c}, \bar{d}, \bar{e}, c, d, e, \bar{f}, \bar{g}, \bar{h}, f, g, h, \bar{z}, z.$$

BACKPROP EXAMPLE

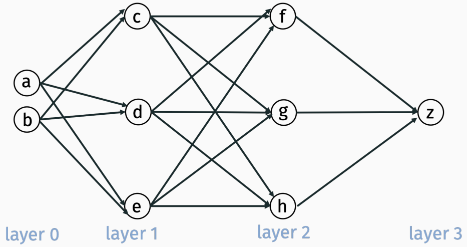
Step 1: Forward pass.



$$\bar{c} = W_{a,c} \cdot a + W_{b,c} \cdot b + \beta_c \qquad c = s(\bar{c})$$
$$\bar{d} = W_{a,d} \cdot a + W_{b,d} \cdot b + \beta_d \qquad d = s(\bar{d})$$
$$\bar{e} = W_{a,e} \cdot a + W_{b,e} \cdot b + \beta_e \qquad e = s(\bar{e})$$
$$\bar{f} = W_{c,f} \cdot c + W_{d,f} \cdot d + W_{e,f} \cdot e + \beta_f \qquad f = s(\bar{f})$$
$$\vdots \qquad \vdots$$
$$\bar{z} = W_{f,z} \cdot f + W_{g,z} \cdot g + W_{h,z} \cdot h + \beta_z \qquad z = s(\bar{z})$$

BACKPROP EXAMPLE

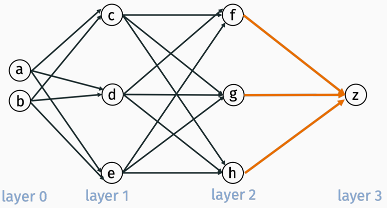
Step 2: Backward pass.



- Using **current parameters** and **computed node values**, compute the partial derivatives of all parameters by moving from right to left.

BACKPROP EXAMPLE

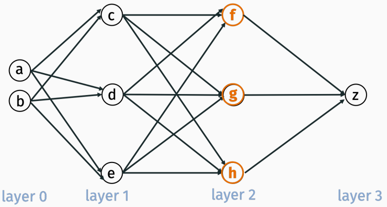
Step 2: Backward pass. Deepest layer.



$$\frac{\partial z}{\partial b_z} = \frac{\partial \bar{z}}{\partial b_z} \cdot \frac{\partial z}{\partial \bar{z}} = 1 \cdot s'(\bar{z})$$
$$\frac{\partial z}{\partial W_{f,z}} = \frac{\partial \bar{z}}{\partial W_{f,z}} \cdot \frac{\partial z}{\partial \bar{z}} = f \cdot s'(\bar{z})$$
$$\frac{\partial z}{\partial W_{g,z}} = \frac{\partial \bar{z}}{\partial W_{g,z}} \cdot \frac{\partial z}{\partial \bar{z}} = g \cdot s'(\bar{z})$$
$$\frac{\partial z}{\partial W_{h,z}} = \frac{\partial \bar{z}}{\partial W_{h,z}} \cdot \frac{\partial z}{\partial \bar{z}} = h \cdot s'(\bar{z})$$

BACKPROP EXAMPLE

Step 2: Backward pass.

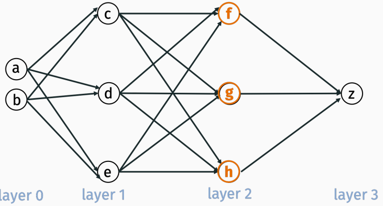


$$\frac{\partial z}{\partial f} = \frac{\partial \bar{z}}{\partial f} \cdot \frac{\partial z}{\partial \bar{z}} = W_{f,z} \cdot s'(\bar{z})$$
$$\frac{\partial z}{\partial g} = \frac{\partial \bar{z}}{\partial g} \cdot \frac{\partial z}{\partial \bar{z}} = W_{g,z} \cdot s'(\bar{z})$$
$$\frac{\partial z}{\partial h} = \frac{\partial \bar{z}}{\partial h} \cdot \frac{\partial z}{\partial \bar{z}} = W_{h,z} \cdot s'(\bar{z})$$

Compute partials with respect to nodes, even though not needed for gradient.

BACKPROP EXAMPLE

Step 2: Backward pass.

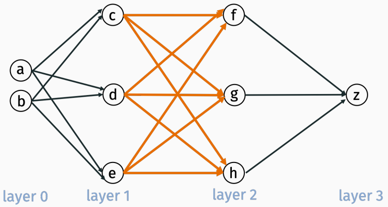


$$\frac{\partial z}{\partial \bar{f}} = \frac{\partial z}{\partial f} \cdot \frac{\partial f}{\partial \bar{f}} = \frac{\partial z}{\partial f} \cdot s'(\bar{f})$$
$$\frac{\partial z}{\partial \bar{g}} = \frac{\partial z}{\partial g} \cdot \frac{\partial g}{\partial \bar{g}} = \frac{\partial z}{\partial g} \cdot s'(\bar{g})$$
$$\frac{\partial z}{\partial \bar{h}} = \frac{\partial z}{\partial h} \cdot \frac{\partial h}{\partial \bar{h}} = \frac{\partial z}{\partial h} \cdot s'(\bar{h})$$

And for nodes pre-nonlinearity

BACKPROP EXAMPLE

Step 2: Backward pass. Next layer.



$$\frac{\partial z}{\partial b_f} = \frac{\partial z}{\partial \bar{f}} \cdot \frac{\partial \bar{f}}{\partial b_f} = \frac{\partial z}{\partial \bar{f}} \cdot 1$$

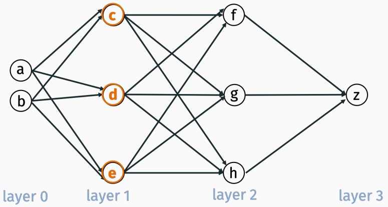
$$\frac{\partial z}{\partial W_{c,f}} = \frac{\partial z}{\partial \bar{f}} \cdot \frac{\partial \bar{f}}{\partial W_{c,f}} = \frac{\partial z}{\partial \bar{f}} \cdot c$$

$$\frac{\partial z}{\partial W_{d,f}} = \frac{\partial z}{\partial \bar{f}} \cdot \frac{\partial \bar{f}}{\partial W_{d,f}} = \frac{\partial z}{\partial \bar{f}} \cdot d$$

$$\frac{\partial z}{\partial W_{e,f}} = \frac{\partial z}{\partial \bar{f}} \cdot \frac{\partial \bar{f}}{\partial W_{e,f}} = \frac{\partial z}{\partial \bar{f}} \cdot e$$

BACKPROP EXAMPLE

Step 2: Backward pass. Next set of nodes.



$$\frac{\partial z}{\partial c} = \frac{\partial z}{\partial \bar{f}} \cdot \frac{\partial \bar{f}}{\partial c} + \frac{\partial z}{\partial \bar{g}} \cdot \frac{\partial \bar{g}}{\partial c} + \frac{\partial z}{\partial \bar{h}} \cdot \frac{\partial \bar{h}}{\partial c}$$

$$\frac{\partial z}{\partial d} = \frac{\partial z}{\partial \bar{f}} \cdot \frac{\partial \bar{f}}{\partial d} + \frac{\partial z}{\partial \bar{g}} \cdot \frac{\partial \bar{g}}{\partial d} + \frac{\partial z}{\partial \bar{h}} \cdot \frac{\partial \bar{h}}{\partial d}$$

$$\frac{\partial z}{\partial e} = \frac{\partial z}{\partial \bar{f}} \cdot \frac{\partial \bar{f}}{\partial e} + \frac{\partial z}{\partial \bar{g}} \cdot \frac{\partial \bar{g}}{\partial e} + \frac{\partial z}{\partial \bar{h}} \cdot \frac{\partial \bar{h}}{\partial e}$$

Multivariate chain rule: Need to sum up impact on gradient from all variables effected in the next layer.

BACKPROP LINEAR ALGEBRA

Linear algebraic view.

Let $\mathbf{v}_i$ be a vector containing the value of all nodes j in layer i.

$$\mathbf{v}_3 = \begin{bmatrix} z \end{bmatrix} \qquad \mathbf{v}_2 = \begin{bmatrix} f \\ g \\ h \end{bmatrix} \qquad \mathbf{v}_1 = \begin{bmatrix} c \\ d \\ e \end{bmatrix}$$

Let $\bar{\mathbf{v}}_i$ be a vector containing $\bar{j}$ for all nodes j in layer i.

$$\bar{\mathbf{v}}_3 = \begin{bmatrix} \bar{z} \end{bmatrix} \qquad \bar{\mathbf{v}}_2 = \begin{bmatrix} \bar{f} \\ \bar{g} \\ \bar{h} \end{bmatrix} \qquad \bar{\mathbf{v}}_1 = \begin{bmatrix} \bar{c} \\ \bar{d} \\ \bar{e} \end{bmatrix}$$

Note: $\mathbf{v}_i = s(\bar{\mathbf{v}}_i)$ where s is applied entrywise.

BACKPROP LINEAR ALGEBRA

Linear algebraic view.

Let $\boldsymbol{\delta}_i$ be a vector containing $\partial z/\partial j$ for all nodes j in layer i.

$$\boldsymbol{\delta}_3 = \begin{bmatrix}1\end{bmatrix} \qquad \boldsymbol{\delta}_2 = \begin{bmatrix}\partial z/\partial f\\ \partial z/\partial g\\ \partial z/\partial h\end{bmatrix} \qquad \boldsymbol{\delta}_1 = \begin{bmatrix}\partial z/\partial c\\ \partial z/\partial d\\ \partial z/\partial e\end{bmatrix}$$

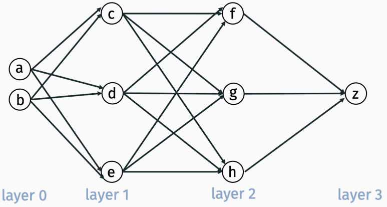
Let $\bar{\boldsymbol{\delta}}_i$ be a vector containing $\partial z/\partial \bar{j}$ for all nodes j in layer i.

$$\bar{\boldsymbol{\delta}}_3 = \begin{bmatrix}\partial z/\partial \bar{z}\end{bmatrix} \qquad \bar{\boldsymbol{\delta}}_2 = \begin{bmatrix}\partial z/\partial \bar{f}\\ \partial z/\partial \bar{g}\\ \partial z/\partial \bar{h}\end{bmatrix} \qquad \bar{\boldsymbol{\delta}}_1 = \begin{bmatrix}\partial z/\partial \bar{c}\\ \partial z/\partial \bar{d}\\ \partial z/\partial \bar{e}\end{bmatrix}$$

Note: $\bar{\boldsymbol{\delta}}_i = s'(\bar{\mathbf{v}}_i) \times \boldsymbol{\delta}_i$ where s' is the derivative of s and this function, as well as the $\times$ are applied entrywise.

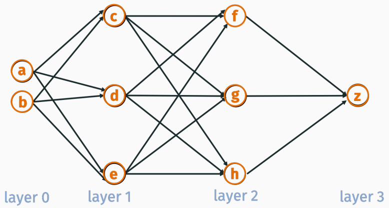
BACKPROP LINEAR ALGEBRA

Let $\mathbf{W}_i$ be a matrix containing all the weights for edges between layer i and layer $i+1$.



$$\mathbf{W}_2 = \begin{bmatrix} W_{f,z} & W_{g,z} & W_{h,z} \end{bmatrix} \quad \mathbf{W}_1 = \begin{bmatrix} W_{c,f} & W_{d,f} & W_{e,f} \\ W_{c,g} & W_{d,g} & W_{e,g} \\ W_{c,h} & W_{d,h} & W_{e,h} \end{bmatrix} \quad \mathbf{W}_0 = \begin{bmatrix} W_{a,c} & W_{b,c} \\ W_{a,d} & W_{b,d} \\ W_{a,e} & W_{b,e} \end{bmatrix}$$

BACKPROP LINEAR ALGEBRA

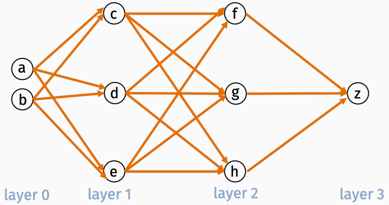


Claim 1: Node derivative computation is matrix multiplication.

$$\vec{\delta}_i = \mathsf{W}_i^T \bar{\delta}_{i+1}$$

BACKPROP LINEAR ALGEBRA

Let $\boldsymbol{\Delta}_i$ be a matrix contain the derivatives for all weights for edges between layer i and layer $i + 1$.

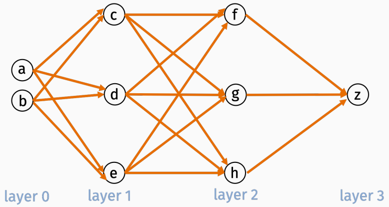


$$\boldsymbol{\Delta}_2 = \begin{bmatrix} \partial z/\partial W_{f,z} & \partial z/\partial W_{g,z} & \partial z/\partial W_{h,z} \end{bmatrix}$$

$$\boldsymbol{\Delta}_1 = \begin{bmatrix} \partial z/\partial W_{c,f} & \partial z/\partial W_{d,f} & \partial z/\partial W_{e,f} \\ \partial z/\partial W_{c,g} & \partial z/\partial W_{d,g} & \partial z/\partial W_{e,g} \\ \partial z/\partial W_{c,h} & \partial z/\partial W_{d,h} & \partial z/\partial W_{e,h} \end{bmatrix}$$

$$\boldsymbol{\Delta}_0 = \ldots$$

BACKPROP LINEAR ALGEBRA



Claim 2: Weight derivative computation is an outer-product.

$$\mathbf{\Delta}_i = \mathsf{v}_i \bar{\delta}_{i+1}^T.$$

BACKPROP

Takeaways:

- Backpropagation can be used to compute derivatives for all weights and biases for any feedforward neural network.
- Final computation boils down to linear algebra operations (matrix multiplication and vector operations) which can be performed quickly on a GPU.

BACKPROP

Backpropagation allows us to compute $\nabla L\left(y_i, f(\vec{\theta}, \vec{x}_i)\right)$ for a single training example $(\vec{x}_i, y_i)$. Computing entire gradient requires computing:

$$\nabla \mathcal{L}(\vec{\theta}) = \sum_{i=1}^{n} \nabla L\left(y_i, f(\vec{\theta}, \vec{x}_i)\right)$$

Computing the entire sum would be very expensive.

$O\left((\text{time for backprop}) \cdot n\right)$ time.

TRAINING NEURAL NETWORKS

Second tool: Stochastic Gradient Descent (SGD).

- Powerful randomized variant of gradient descent used to train neural networks.
- Or any other model where computing gradients is expensive.

Recall gradient descent update:

- For $t = 1, \ldots, T$:
    - $\vec{\theta}_{t+1} = \vec{\theta}_t - \eta \nabla \mathcal{L}(\vec{\theta}_t)$

where η is a learning rate parameter.

STOCHASTIC GRADIENT DESCENT

Let $L_j(\vec{\theta})$ denote $L\left(y_j, f(\vec{\theta}, \vec{x}_j)\right)$.

Claim: If $j \in 1, \ldots, n$ is chosen uniformly at random. Then:

$$n \cdot \mathbb{E}\left[\nabla L_j(\vec{\theta})\right] = \nabla \mathcal{L}(\vec{\theta}).$$

$\nabla L_j(\vec{\theta})$ is called a stochastic gradient.

STOCHASTIC GRADIENT DESCENT

SGD iteration:

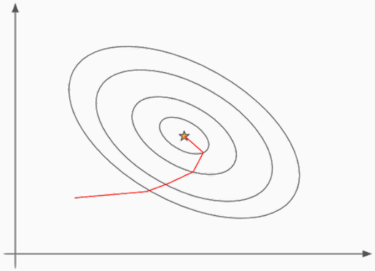
- Initialize $\vec{\theta}_0$ (typically randomly).
- For $t = 1, \ldots, T$:
    - Choose j uniformly at random.
    - Compute stochastic gradient $\vec{g} = \nabla L_j(\vec{\theta}_t)$.
        - For neural networks this is done using backprop with training example $(\vec{x}_j, y_j)$.
    - Update $\vec{\theta}_{t+1} = \vec{\theta}_t - \eta\vec{g}$

Move in direction of steepest descent in expectation.

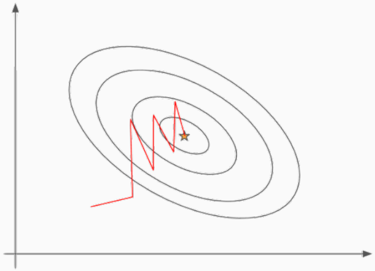
STOCHASTIC GRADIENT DESCENT

Gradient descent: Fewer iterations to converge, higher cost per iteration.

Stochastic Gradient descent: More iterations to converge, lower cost per iteration.



Gradient Descent

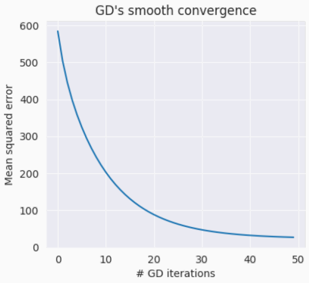


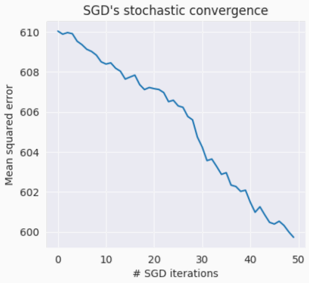
Stochastic Gradient Descent

STOCHASTIC GRADIENT DESCENT

Gradient descent: Fewer iterations to converge, higher cost per iteration.

Stochastic Gradient descent: More iterations to converge, lower cost per iteration.





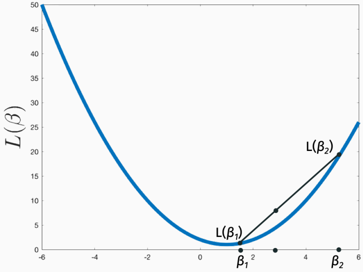
CONVERGENCE

Like standard gradient descent, stochastic gradient descent is only guaranteed to converge to the minimizer of a convex loss function.

Definition (Convex)

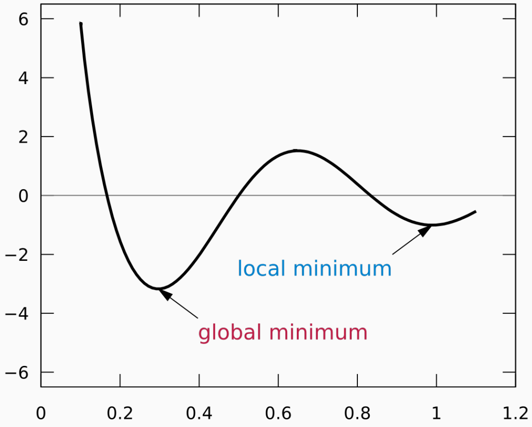
A function L is convex iff for any $\vec{\beta}_1, \vec{\beta}_2, \lambda \in [0,1]$:

$$(1-\lambda)\cdot L(\vec{\beta}_1) + \lambda \cdot L(\vec{\beta}_2) \geq L\left((1-\lambda)\cdot \vec{\beta}_1 + \lambda \cdot \vec{\beta}_2\right)$$



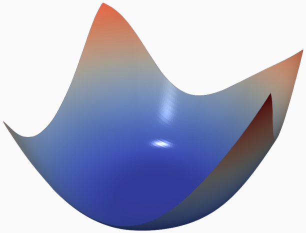
CONVERGENCE OF GRADIENT DESCENT

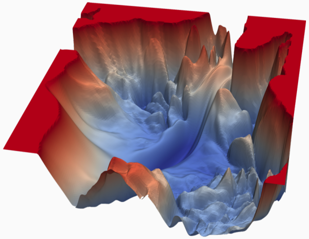
Without convexity, we can only expect to converge to a local minimum.



CONVERGENCE

Least squares regression, logistic regression, SVMs, even all of these with kernels lead to convex losses.


convex loss

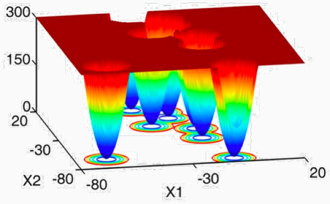

cross-entropy loss for neural net

Neural networks very much do not...

CONVERGENCE

But SGD still performs remarkably well in practice. Understanding this phenomenon is a major open research question in machine learning and soptimization. Current hypotheses include:

- Initialization seems important (random uniform vs. random Gaussian vs. Xavier initialization vs. He initialization vs. etc.)
- Randomization helps in escaping local minima.
- All local minima are global minima?
- SGD finds "good" local minima?



STOCHASTIC GRADIENT DESCENT IN PRACTICE

Practical Modification 1: **Shuffled Gradient Descent.**

Instead of choosing j randomly at each iteration, randomly permute (shuffle) data and set $j = 1, \ldots, n$. After every n iterations, reshuffle data and repeat.

Question: Why might we want to do this?

STOCHASTIC GRADIENT DESCENT IN PRACTICE

Practical Modification 1: **Shuffled Gradient Descent.**

Instead of choosing j randomly at each iteration, randomly permute (shuffle) data and set $j = 1, \ldots, n$. After every n iterations, reshuffle data and repeat.

- Relatively similar convergence behavior to standard SGD.
- **Important term:** one epoch denotes one pass over all training examples: $j = 1, \ldots, j = n$.
- Convergence rates for training neural networks are often discussed in terms of epochs instead of iterations.

STOCHASTIC GRADIENT DESCENT IN PRACTICE

Practical Modification 2: **Mini-batch Gradient Descent.**

Observe that for any batch size s,

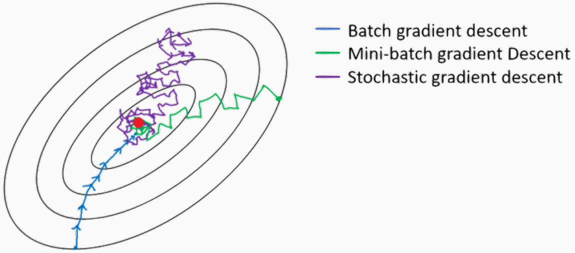
$$n \cdot \mathbb{E}\left[\frac{1}{s}\sum_{i=1}^{s} \nabla L_{j_i}(\vec{\theta})\right] = \nabla \mathcal{L}(\vec{\theta}).$$

if $j_1, \ldots, j_s$ are chosen independently and uniformly at random from $1, \ldots, n$.

Instead of computing a full stochastic gradient, compute the average gradient of a small random set (a mini-batch) of training data examples.

Question: Why might we want to do this?

MINI-BATCH GRADIENT DESCENT



- For small batch size s, mini-batch gradients are nearly as fast to compute as stochastic gradients (due to parallelism).
- Overall faster convergence (fewer iterations needed).

STOCHASTIC GRADIENT DESCENT IN PRACTICE

Practical Mod. 3: **Per-parameter adaptive learning rate.**

Let $\vec{g} = \begin{bmatrix} g_1 \\ \vdots \\ g_p \end{bmatrix}$ be a stochastic or batch stochastic gradient. Our typical parameter update looks like:

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \eta \vec{g}.$$

We've already seen a simple method for adaptively choosing the learning rate/step size η. Worked well for convex functions.

STOCHASTIC GRADIENT DESCENT IN PRACTICE

Practical Mod. 3: **Per-parameter adaptive learning rate.**

In practice, neural networks can often be optimized much faster by using "adaptive gradient methods" like Adagrad, Adadelta, RMSProp, and ADAM. These methods make updates of the form:

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \begin{bmatrix} \eta_1 \cdot g_1 \\ \vdots \\ \eta_p \cdot g_p \end{bmatrix}$$

So we have a separate learning rate for each entry in the gradient (e.g. parameter in the model). And each $\eta_1, \ldots, \eta_p$ is chosen adaptively.

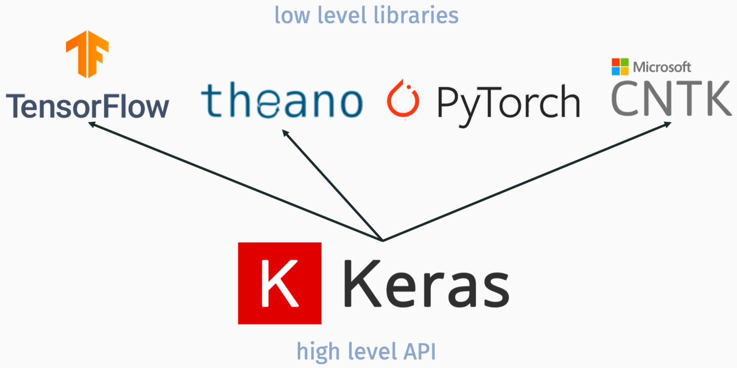
NEURAL NETWORK DEMOS

Two demos uploaded on neural networks:

- `keras_demo_synthetic.ipynb`
- `keras_demo_mnist.ipynb`

Please spend some time working through these!

NEURAL NETWORK SOFTWARE

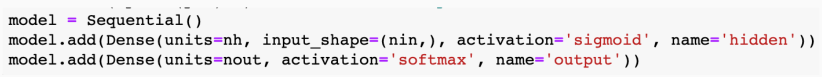


Low-level libraries have built in optimizers (SGD and improvements) and can automatically perform backpropagation for arbitrary network structures. Also ptimize code for any available GPUs.

Keras has high level functions for defining and training a neural network architecture.

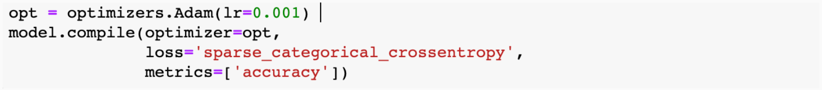
NEURAL NETWORK SOFTWARE

Define model:



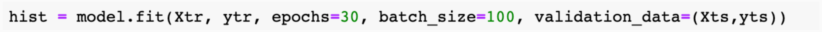
```
model = Sequential()
model.add(Dense(units=nh, input_shape=(nin,), activation='sigmoid', name='hidden'))
model.add(Dense(units=nout, activation='softmax', name='output'))
```

Compile model:



```
opt = optimizers.Adam(lr=0.001)
model.compile(optimizer=opt,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

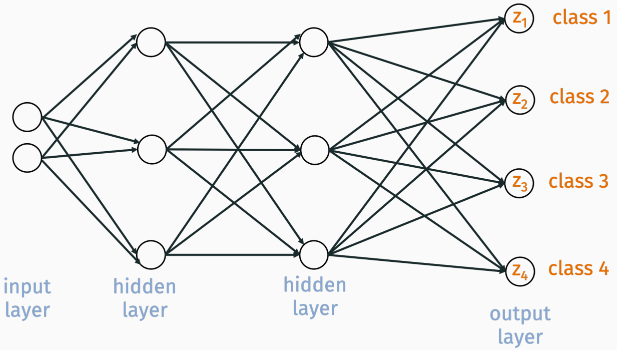
Train model:



```
hist = model.fit(Xtr, ytr, epochs=30, batch_size=100, validation_data=(Xts,yts))
```

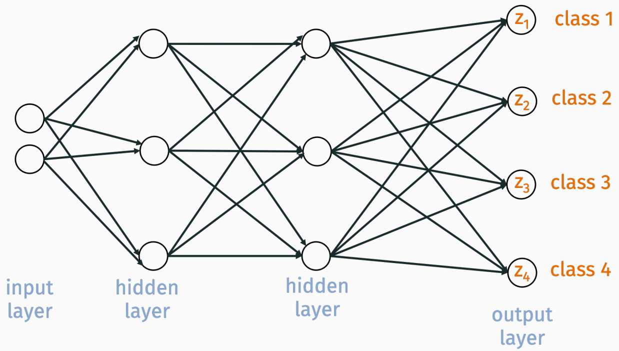
MULTICLASS CLASSIFICATION

The MNIST lab performs multiclass classification. Typically approach to multiclass problems with neural networks is to have one output neuron per class:



Classification rule: Place in input $\vec{x}$ in class i if z_i is the neuron with maximum value after running $\vec{x}$ through the network.

MULTICLASS CLASSIFICATION



Last layer typically uses a "softmax" nonlinearity to map all values $\bar{z}_1, \ldots, \bar{z}_q$ to values between 0 and 1:

$$z_i = \frac{e^{-\bar{z}_i}}{\sum_{j=1}^{q} e^{-\bar{z}_j}}.$$

MULTICLASS CLASSIFICATION

Trained using multiclass cross-entropy loss. Let $z_1(\vec{x}, \theta), \ldots, z_q(\vec{x}, \theta)$ be the outputs obtain when running the network on input $\vec{x}$ with parameters (weights and baises) $\vec{\theta}$.

$$L(y, \vec{x}, \vec{\theta}) = -\sum_{i=1}^{q} \mathbb{1}[y = i] \log(z_i(\vec{x}, \theta)).$$

Overall loss for training data $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)$ is:

$$\mathcal{L}(\vec{\theta}) = \sum_{i=1}^{n} L(y_i, \vec{x}_i, \vec{\theta})$$

Used in our demo and very standard for neural network classification.