Optimization for Deep Models

Initialization, Activation, and Normalization

N. Rich Nguyen, PhD

SYS 6016

0. Challenges

Optimization in ML

Optimization: finding the parameters θ of a model that significantly reduce a cost function $J(\Theta)$, which typically includes a performance measure evaluated on the entire training set as well as additional regularization terms.

The simplest way to convert a ML problem back into an optimization problem is to minimize the expected lost on the training set → **empirical risk minimization**

$$\mathbb{E}_{\mathbf{x},y}[L(f(\mathbf{x};\theta),\mathbf{y})] = \frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)};\theta),y^{(i)})$$

Rather than optimizing the **risk directly**, we optimize the empirical risk and hope that the risk decreases significantly as well.

Optimization for Neural Networks

Unfortunately, the empirical risk minimization is prone to **overfitting** as model with **high capacity** such as a neural net can simply memorize the training set.

In deep learning, we use a slightly different approach here → **mini-batch stochastic methods**

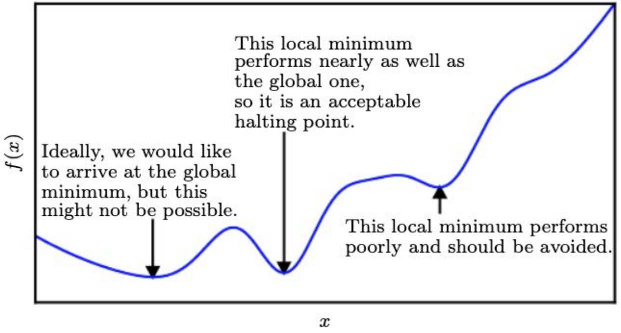
- Mini-batch sizes are driven by hardware, parallel, memory. It is common for power of 2 as batch sizes (32--256)
- It is crucial that the mini-batch is selected **randomly** → **stochastic**

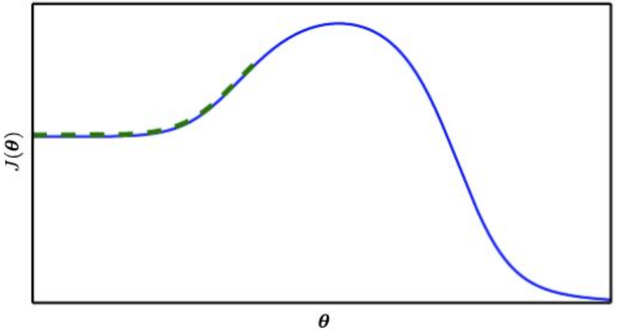
Compute the gradient of the loss w.r.t. the parameters for that minibatch, then updating the parameters in the direction of the gradient:

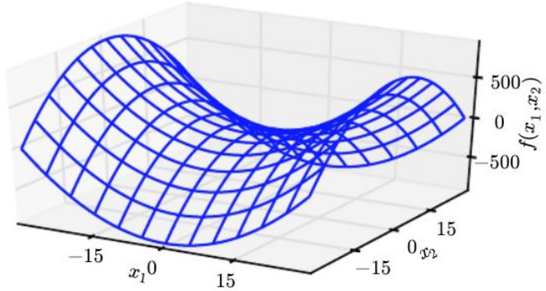
$$\mathbf{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(\mathbf{x}^{(i)}; \theta), y^{(i)})$$

$$\theta \leftarrow \theta - \epsilon \mathbf{g}$$

Local Minima, Plateaus, and Saddle Points







- With non-convex functions, such as neural nets, it’s possible to have many **local minima**
- Local minima with high cost could pose a serious problem for gradient-based algo
- High-D non-convex functions, another point with zero gradient: **a saddle point**
- Some points around a saddle point have **greater** cost while others have **lower** cost

Training Deep Models

To tackle complex problems such as a visual perception task, we usually need to train a much **deeper network** with hundreds of layers and thousands of neurons.

A few issue with deep models includes:

- Training will be extremely slow
- Millions of parameters risking overfitting
- Facing the tricky “vanishing gradients” problem?

The Vanishing Gradient Problem

- Backpropagation works by passing the **error gradient** back and forth among input, output, and hidden layers. It uses gradient to update the parameters
- For **deep** computational graph, repeated applying the same operation at each time step makes the gradients get **smaller and smaller** to which point they leave the weights virtually **unchanged** → **vanishing gradient problem**

```
0.5^100 =
7.8886091e-31
```

The Exploding Gradient Problem

- The opposite of vanishing gradient can also happen
- **Repeated** matrix multiplication at each time step of a computational graph
- Gradients grow bigger and bigger → many layers got **insanely large** weight updates, and the network becomes unstable and diverged

```
1.5^100 =
4.0656118e+17
```

A reason why deep neural networks were abandoned for 2 decades!

Optimization of Deep Neural Nets

1. **Initialization:** how to initialize the weights so that they do not saturate?
2. **Activation:** how to solve the vanishing gradient problem?
3. **Normalization:** how to get the model to learn the optimal scale?
4. **Optimizers:** when gradient descent was too slow or not good enough?
5. **Adaptive Learning Rate**: what if convergence is too slow or sub-optimal?
6. **Second-Order Training Methods**: can we make use of second derivatives?

1. Initialization

Xavier Initialization

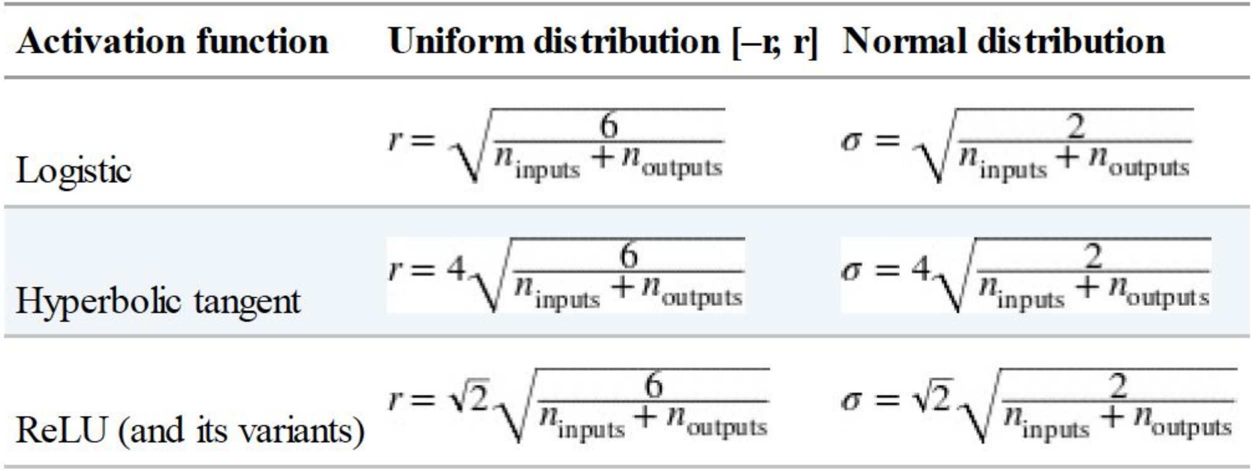
- Iterative algorithms require the user to specify some initial point from which to begin the iterations → deep learning are **strongly** affected by **initialization**
- We don’t want the signal to **vanish**, **explode**, nor **saturate** → break **symmetry** between different units.
- To keep it **flow symmetrically,** Xavier Glorot and Yoshua Bengio argue that “*we need the variance of the outputs of each layer to be **equal** to the variance of its inputs*” → Xavier Initialization for Logistic Activation (2010)
- This heuristic strategy led to the current success of Deep Learning.

$$\mathbf{W}_{i,j} \sim \text{Uniform}\left(-\sqrt{\frac{6}{n_{\text{inputs}}+n_{\text{outputs}}}}, \sqrt{\frac{6}{n_{\text{inputs}}+n_{\text{outputs}}}}\right)$$

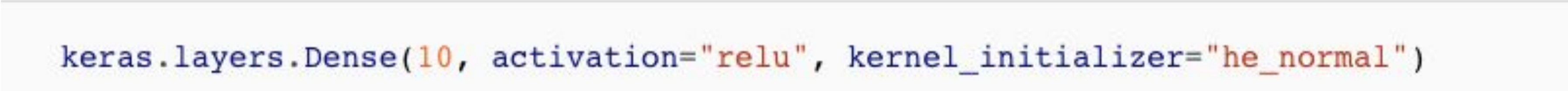
$$\mathbf{W}_{i,j} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{\text{inputs}}+n_{\text{outputs}}}}\right)$$

He Initialization

Similar to Xavier Initialization, but for **different activation functions**



Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

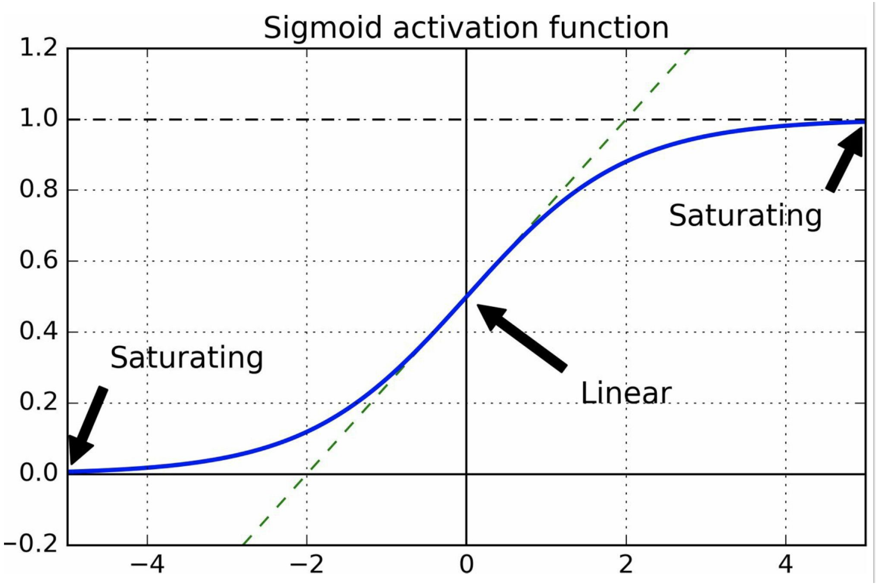


```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

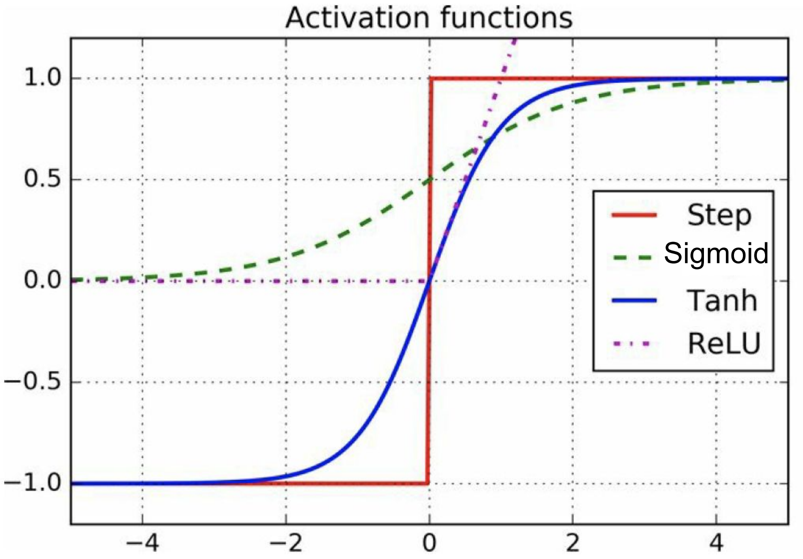
2. Activation

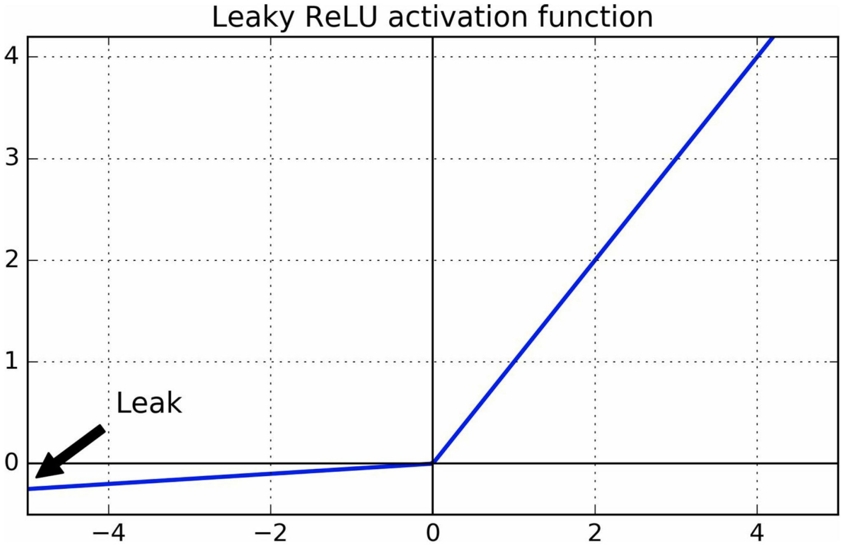
Activation Function Choices

- Poor choice of activation function can lead to vanishing/exploding gradient
- **Nature** chooses to use roughly **sigmoid activation** function in biological neurons, but it turns out that other functions (eg. ReLU) behave **much faster and better.**



ReLU Activation



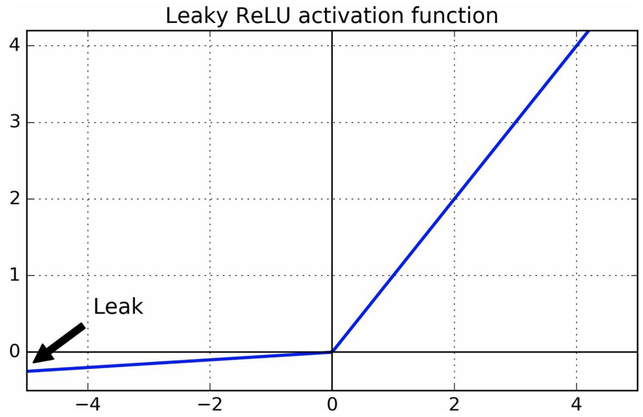


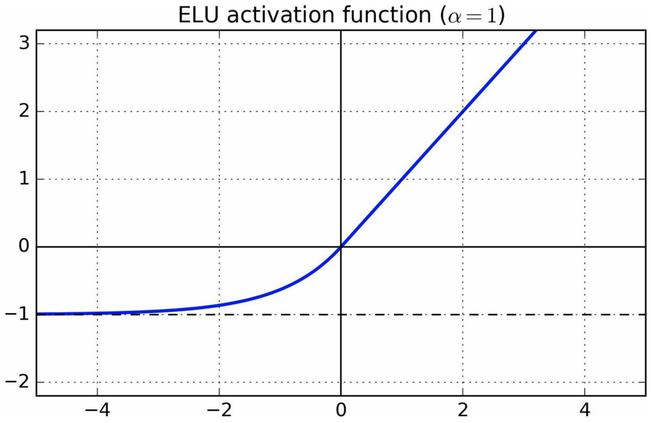
- Fast to compute, but suffer from **dying**: meaning not outputting anything but 0
- To solve this, you can use a variant of ReLU called leaky ReLU

ELU (Exponential Linear Unit)

Outperform ReLU on faster convergence and accuracy (Clevert et al 2015)

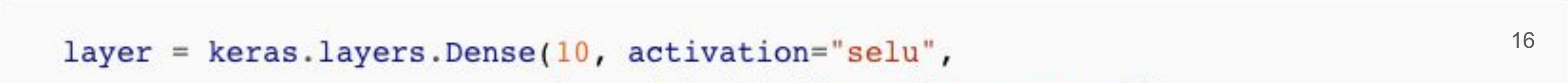
$$\mathrm{ELU}_{\alpha}(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & ifz \geq 0 \end{cases}$$





Why ELU is better than ReLU?

Dying neuron? non-zero gradient? smoothness?



```
layer = keras.layers.Dense(10, activation="selu",
```

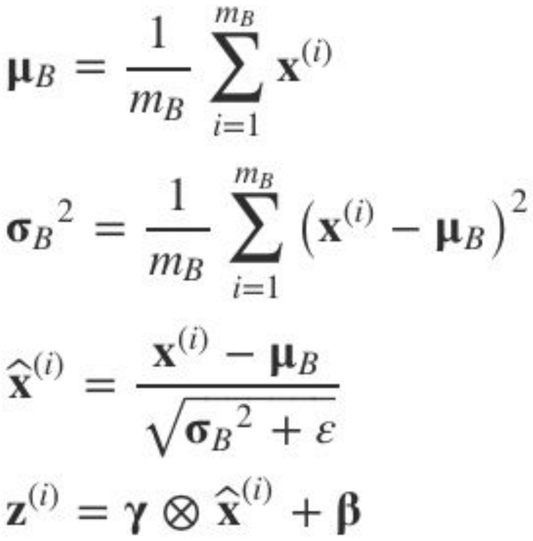
3. Normalization

Batch Normalization

- He Initialization reduces vanishing/exploding gradient problems at the **beginning** of training, but does not guarantee they won’t come back **during** training
- Sergey Ioffe and Christian Szegedy (2015) address this by a technique called Batch Normalization (BN)
- This adds an operation **before** activation function: simply **zero-centering** and **normalizing** the inputs, then **scaling** and **shifting** the results → optimal scale.

Batch Normalization

Learn 4 parameters: **scale**, **offset**, **mean**, and **standard deviation**

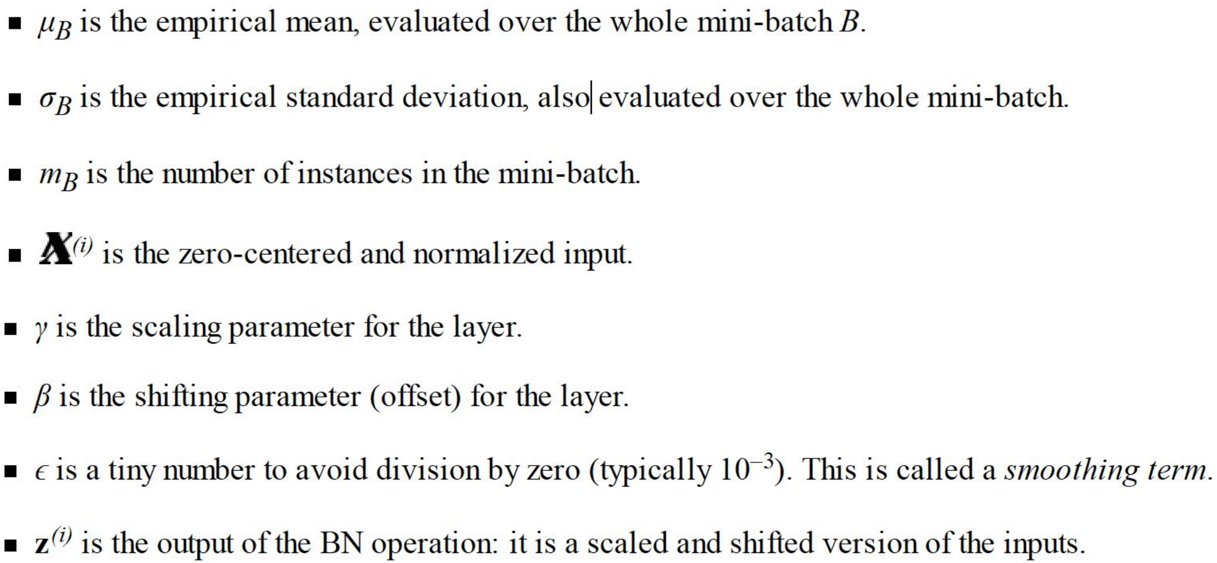


$$\boldsymbol{\mu}_B = \frac{1}{m_B}\sum_{i=1}^{m_B}\mathbf{x}^{(i)}$$

$${\boldsymbol{\sigma}_B}^2 = \frac{1}{m_B}\sum_{i=1}^{m_B}\left(\mathbf{x}^{(i)} - \boldsymbol{\mu}_B\right)^2$$

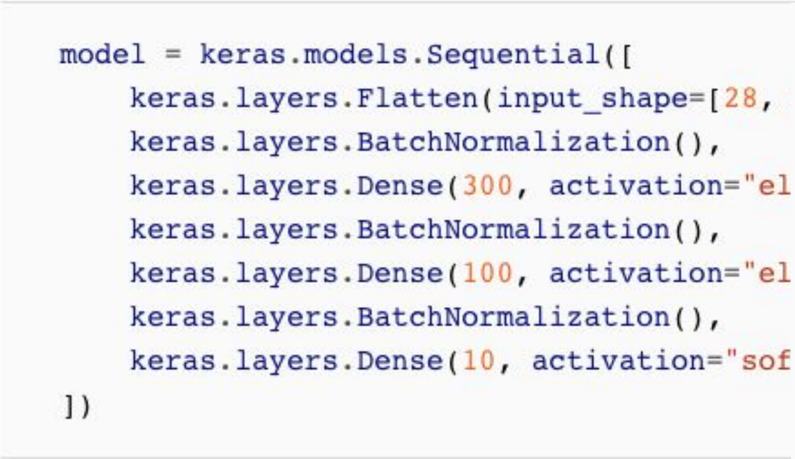
$$\widehat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{{\boldsymbol{\sigma}_B}^2 + \varepsilon}}$$

$$\mathbf{z}^{(i)} = \boldsymbol{\gamma} \otimes \widehat{\mathbf{x}}^{(i)} + \boldsymbol{\beta}$$

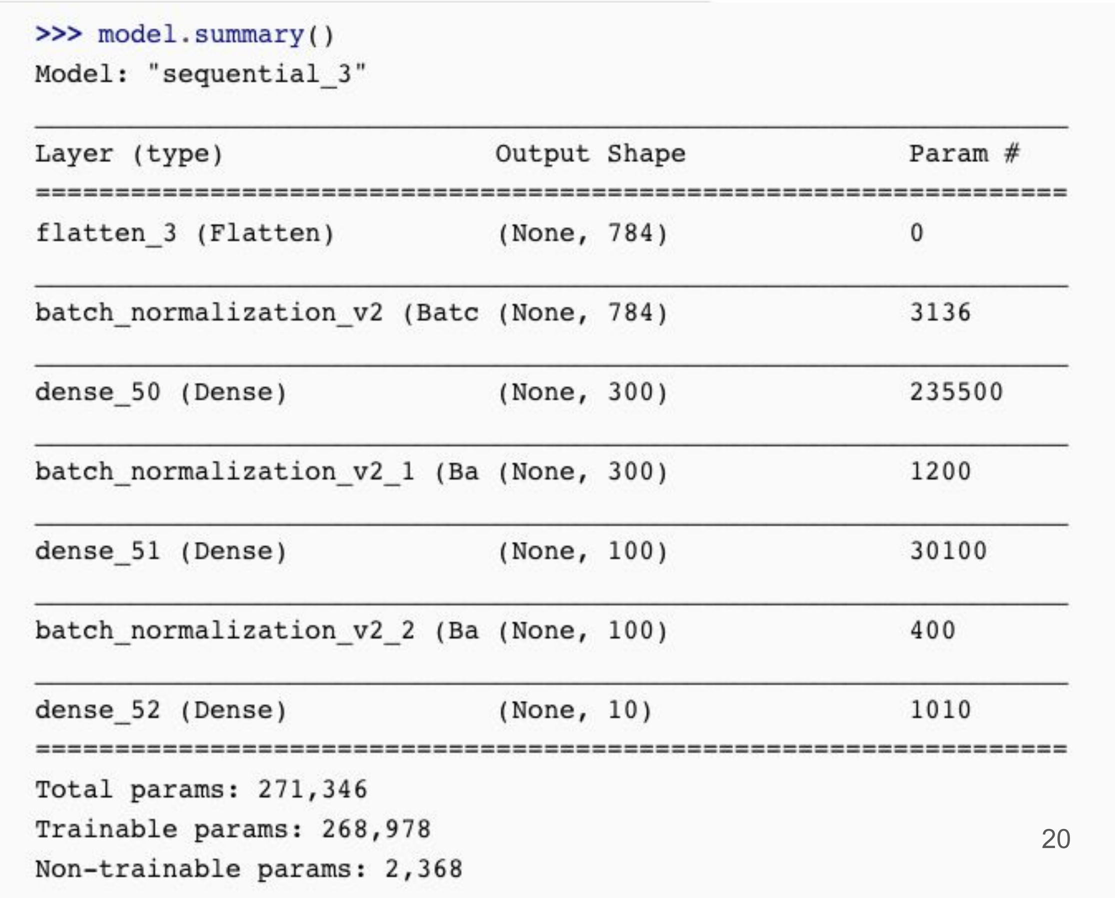


- μ_B is the empirical mean, evaluated over the whole mini-batch B.
- σ_B is the empirical standard deviation, also evaluated over the whole mini-batch.
- m_B is the number of instances in the mini-batch.
- $\hat{\mathbf{x}}^{(i)}$ is the zero-centered and normalized input.
- γ is the scaling parameter for the layer.
- β is the shifting parameter (offset) for the layer.
- ϵ is a tiny number to avoid division by zero (typically 10^{-3}). This is called a *smoothing term*.
- $\mathbf{z}^{(i)}$ is the output of the BN operation: it is a scaled and shifted version of the inputs.

Batch Normalization Implementation



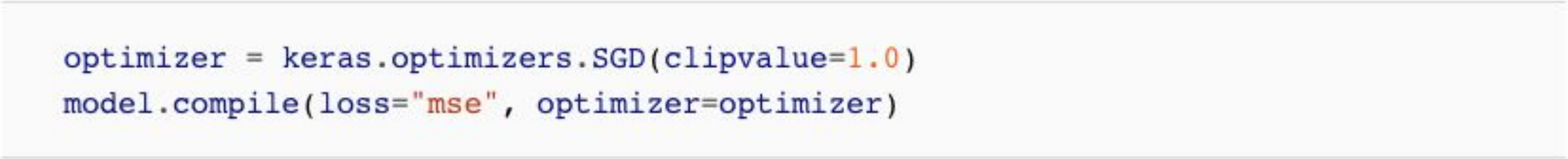
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28,
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="el
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="el
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="sof
])
```



```
>>> model.summary()
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_3 (Flatten)          (None, 784)               0
_________________________________________________________________
batch_normalization_v2 (Batc (None, 784)               3136
_________________________________________________________________
dense_50 (Dense)             (None, 300)               235500
_________________________________________________________________
batch_normalization_v2_1 (Ba (None, 300)               1200
_________________________________________________________________
dense_51 (Dense)             (None, 100)               30100
_________________________________________________________________
batch_normalization_v2_2 (Ba (None, 100)               400
_________________________________________________________________
dense_52 (Dense)             (None, 10)                1010
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
```

Gradient Clipping

- Use as an alternative to Batch Normalization
- **Quick and dirty:** simply clip the gradients during backpropagation so that they can never exceed some threshold.
- Often use in Recurrent Neural Nets (RNNs)



```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

Recap: Optimization for Deep Neural Nets

1. **Initialization:** how to initialize the weights so that they do not saturate?
2. **Activation:** how to solve the vanishing gradient problem?
3. **Normalization:** how to get the model to learn the optimal scale?
4. **Optimizers:** when gradient descent was too slow or not good enough?
5. **Adaptive Learning Rate**: what if convergence is too slow or sub-optimal?
6. **Second-Order Training Methods**: can we make use of second derivatives?