# Hacks to Build ANN Models

Instructor: Revendranath T

## Agenda

1. Number of layers in neural network model
2. Number of neurons per layer
3. Avoiding Vanishing/Exploding gradients problem
4. Weights initialization
5. Selecting Optimizers
6. Selecting learning rates
7. Batch Size
8. Batch Normalization
9. Avoid over-fitting through regularization

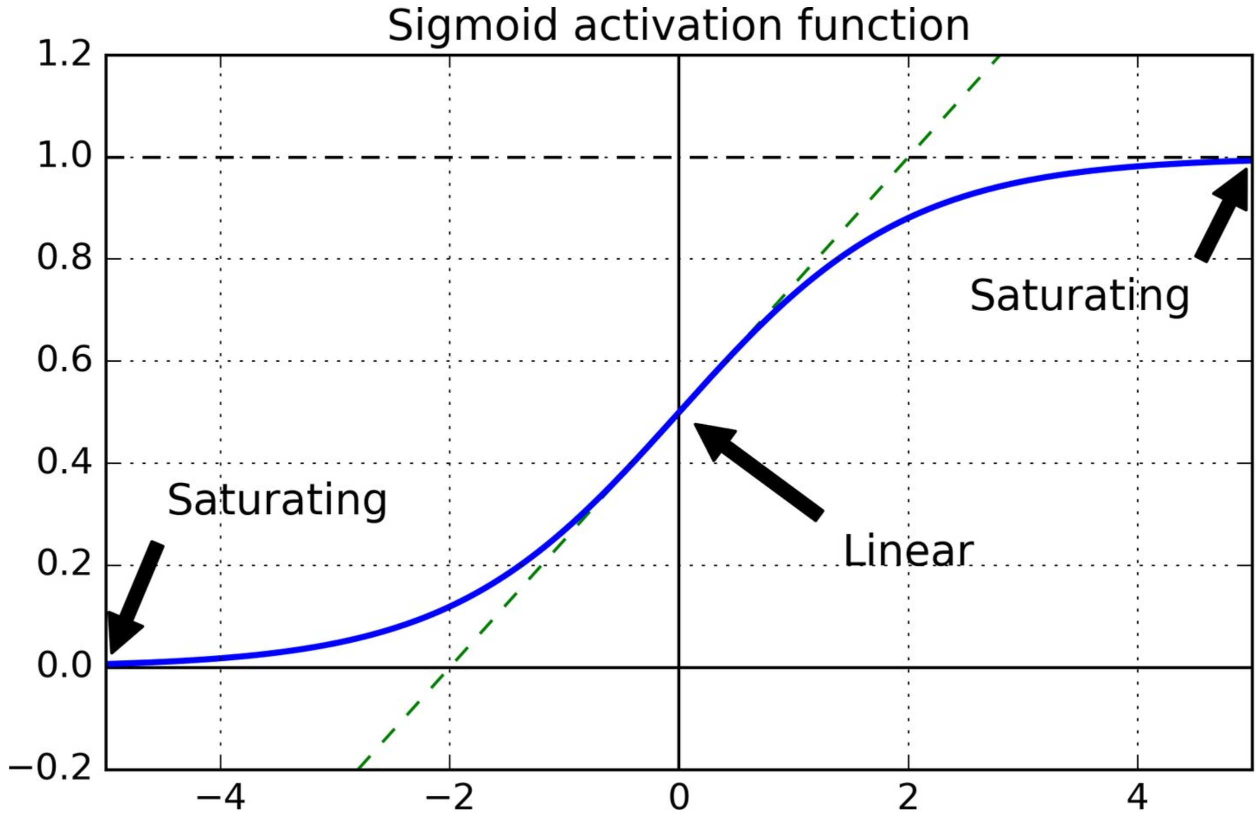
## 1. Number of layers in neural network model

1. Any neural network with one or two hidden layers will work
2. Complex tasks with huge volumes of data requires dozens of hidden layers.
3. One trick is to increase the number of hidden layers till the neural network model starts over-fitting the training data
4. After the model over-fits the training data, reduce some layers or neurons in layers using regularization techniques

## 2. Number of neurons per layer

1. Number of neurons for input and output layers are decided by the type of data.
2. For example, if you are classifying a 28x28 images into living and non-living objects, then input layers consists of 784 (28x28) neurons and output layers has 2 neurons
3. Suggested that neurons in hidden layers form an inverted pyramid from the input layers to the output layers
   a. Say a NN model has 300 neurons at a hidden layer next to input layer, the next layer should have less than 300, say 200, and the next layer should have less than 200, say 100, and continue to reduce neurons till the output layer
   b. In some scenarios using same number of neurons in all hidden layers is practiced.
4. Continue to increase the number of neurons per layer until the model is over-fitting, and then perform regularization.
5. Prefer to have more layers than more number of neurons per layer so that bottlenecks due to many neurons in one layer can be avoided.

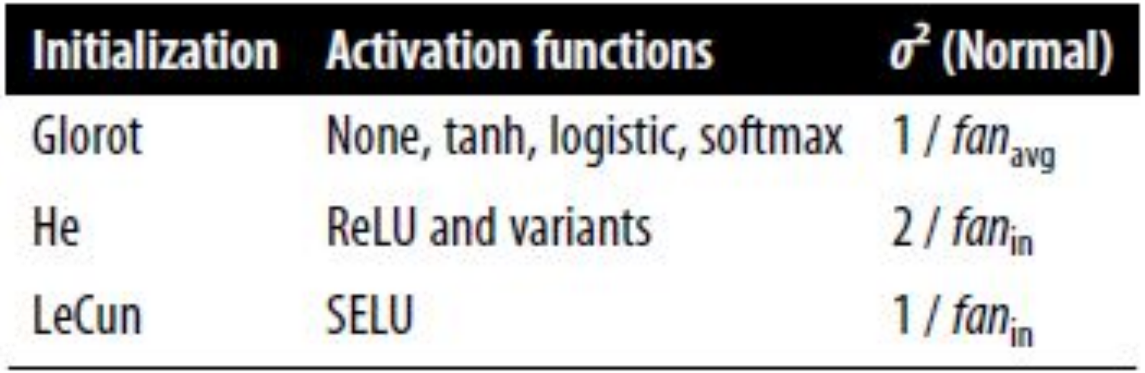
# 3. Avoiding Vanishing Gradient Problem



## 3. Avoiding Vanishing Gradient Problem

- Gradients often get smaller and smaller as the algorithm progresses down to the lower layers.
- As a result, the Gradient Descent update leaves the lower layers' connection weights virtually unchanged, and training never converges to a good solution.
- This is the vanishing gradients problem.
- In some cases, the opposite can happen: the gradients can grow bigger and bigger until layers get insanely large weight updates and the algorithm diverges.
- This is the exploding gradients problem, which surfaces in recurrent neural networks
- Two approaches to solve this problem:
    - Weights initialization before training the model &
    - Use non-saturating activation functions such as ELU or SELU

## 4. Weights Initialization



| Initialization | Activation functions | $\sigma^2$ (Normal) |
|---|---|---|
| Glorot | None, tanh, logistic, softmax | $1 / fan_{avg}$ |
| He | ReLU and variants | $2 / fan_{in}$ |
| LeCun | SELU | $1 / fan_{in}$ |

1. Code Illustration:

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

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

## 5. Optimizers

1. Adam or Nadam
2. If data is new, and models do not train quickly, or if error is high after using Adam or Nadam, then toy with other optimizers.

## 6. Learning Rate

1. Maximum learning rate is the learning rate above which the training algorithm diverges
2. Optimal learning rate is about half of the maximum learning rate
3. Approach to find optimal learning rate:
    a. Train the model for a few hundred iterations with low learning rate and gradually increase the learning rate by multiplying with a constant value
    b. Plot loss function against learning rate, the loss initially decreases and suddenly increases
    c. Optimal learning rate is the point at which loss again begins to increase
4. Preferred methods to find learning rates
    a. Exponential scheduling
    b. 1cycle scheduling (performs better than all others)

## 7. Batch Size

1. Large batch sizes may lead to training instabilities
2. Suggested batch size range between 2 to 32 in multiples of 2.
3. Large batch sizes are possible when learning rates are warmed up i.e., start with a small learning rate and ramp it up.
    a. This approach lead to short training time

## 8. Batch Normalization

1. Vanishing or Exploding Gradient problem may not be solely eliminated by initialization of weights and ELU activation function.
2. Along with initialization and ELU activation functions, Batch Normalization improves chances of reducing the occurrence of vanishing or exploding gradient problem
3. Batch Normalization technique adds normalized values of inputs after activation functions.

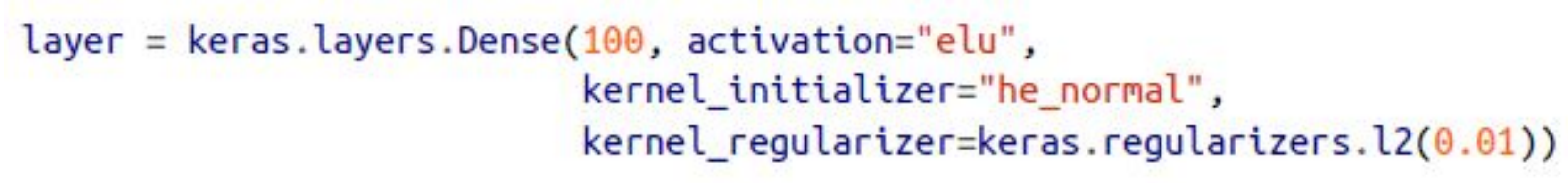
## 9. Regularization

1. $l_1$ and $l_2$ regularization
2. Dropout
3. Monte Carlo (MC) Dropout
4. Max-Norm regularization

## 9. Regularization

1. $l_1$ and $l_2$ regularization
    a. Use $\ell_2$ regularization to constrain a neural network's connection weights, and/or
    b. Error is calculated in each training step and added to the final loss
    c. Use $\ell_1$ regularization if you want a sparse model (with many weights equal to 0)

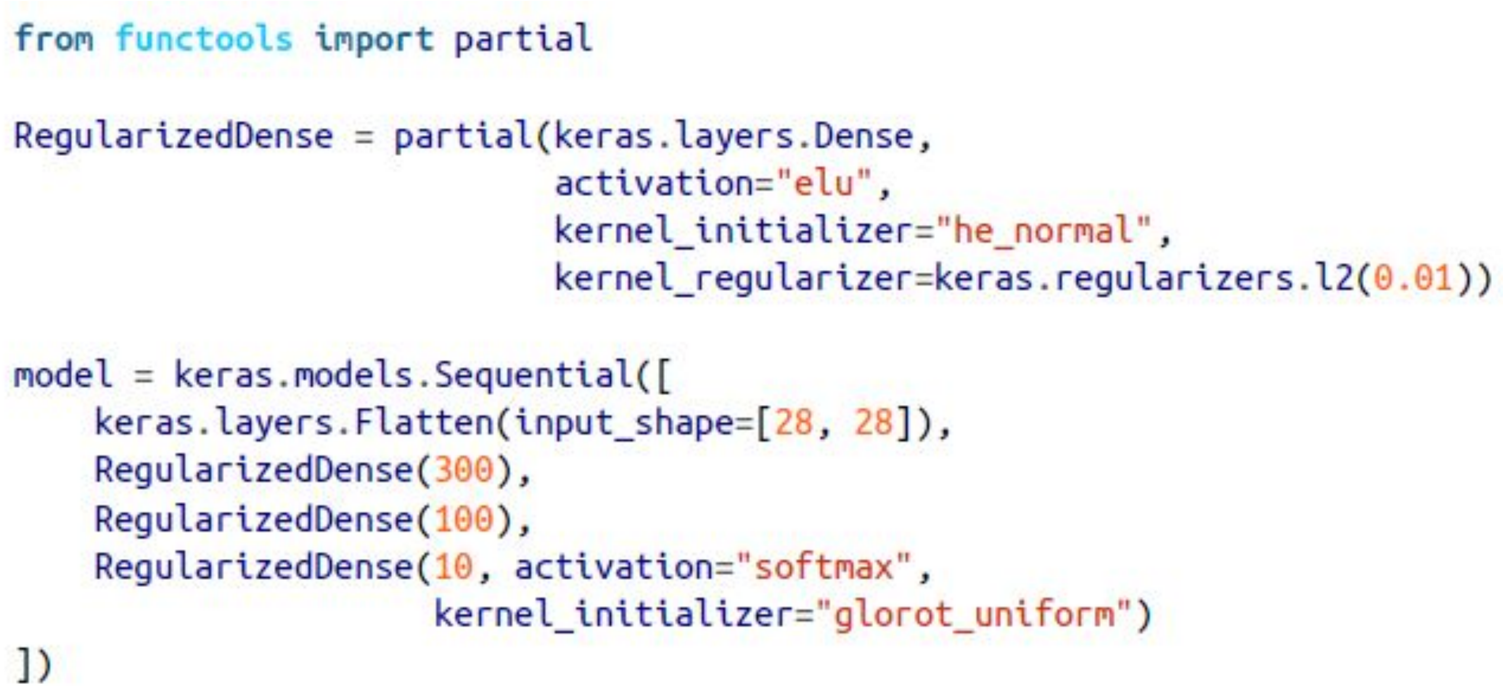
2. Implementation in Keras



```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
```

## 9. Regularization

$l_1$ and $l_2$ regularization implementation in Keras using partial in Python’s functools.partial()

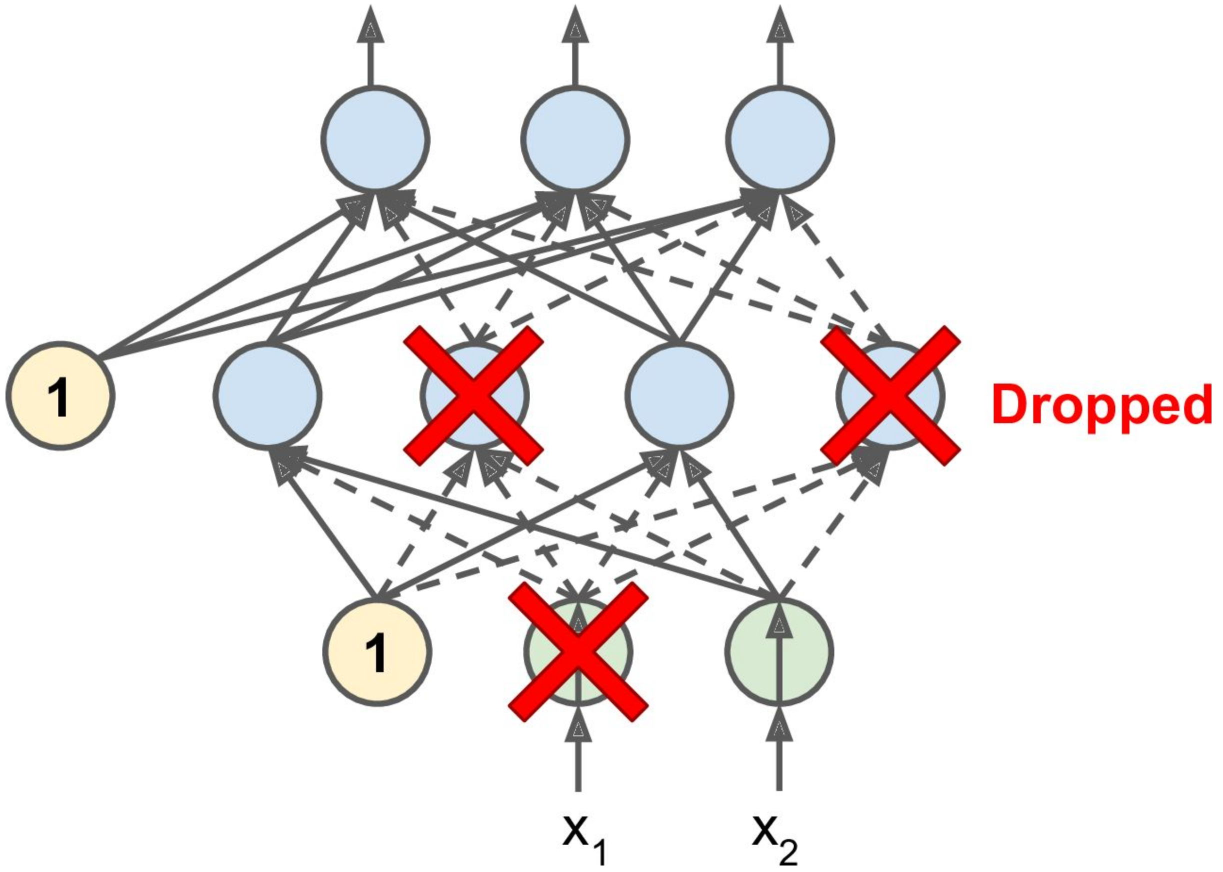


```python
from functools import partial

RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax",
                     kernel_initializer="glorot_uniform")
])
```

## 9. Regularization: Dropout

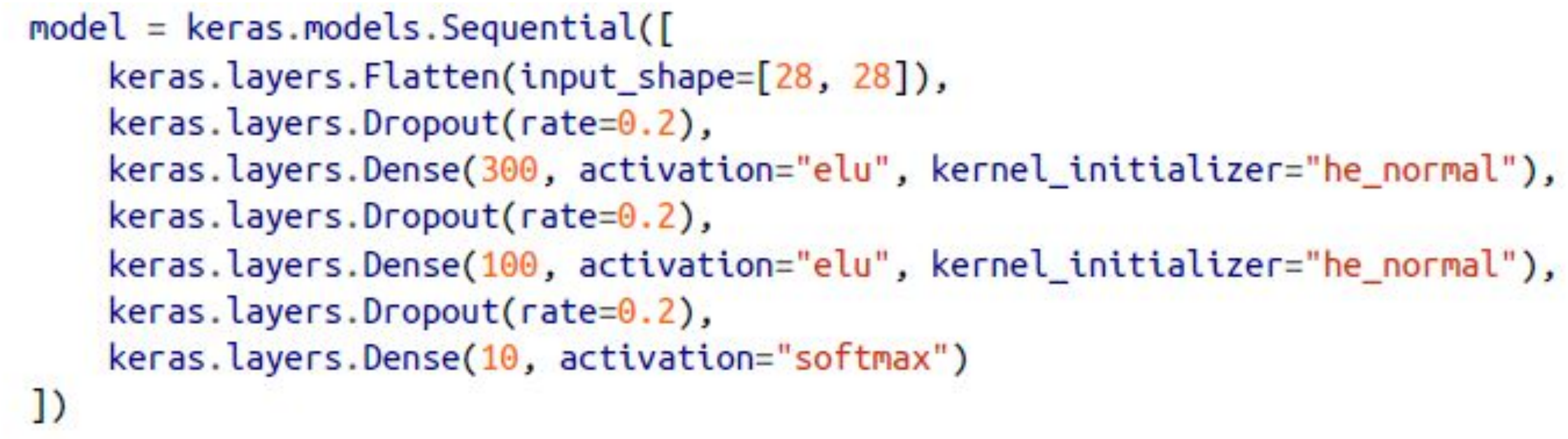


## 9. Regularization

- Dropout: at every training step, every neuron has a probability **p** of being temporarily “dropped out,” meaning it will be entirely ignored during this training step, but it may be active during the next step.
- Includes the input neurons, but always excluding the output neurons.
- The hyperparameter **p** is called the dropout rate,
  - typically set between 10% and 50%:
  - closer to 20–30% in recurrent neural nets, and
  - closer to 40–50% in convolutional neural networks.
- After training, neurons don’t get dropped anymore.

## 9. Regularization

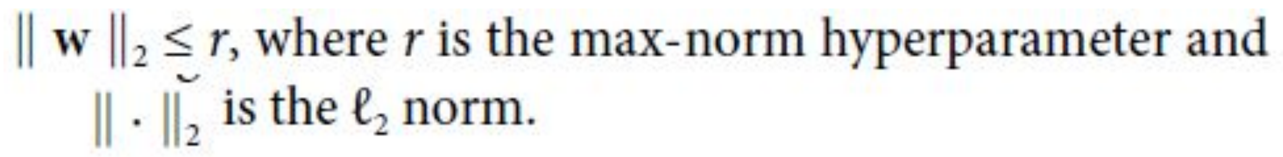
- Dropout: Implementation in Keras



```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

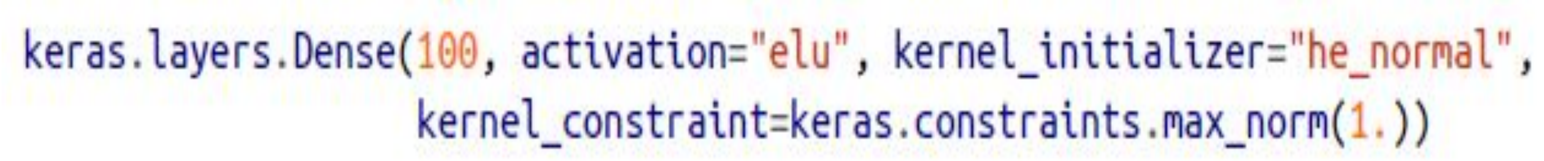
# 9. Regularization

- **Monte Carlo Dropout:**
- Boosts the performance of any trained dropout model without having to retrain it or even modify it at all,
- Provides a much better measure of the model’s uncertainty, and is also amazingly simple to implement.
- **Max-Norm Regularization:**
- For each neuron, it constrains the weights w of the incoming connections such that



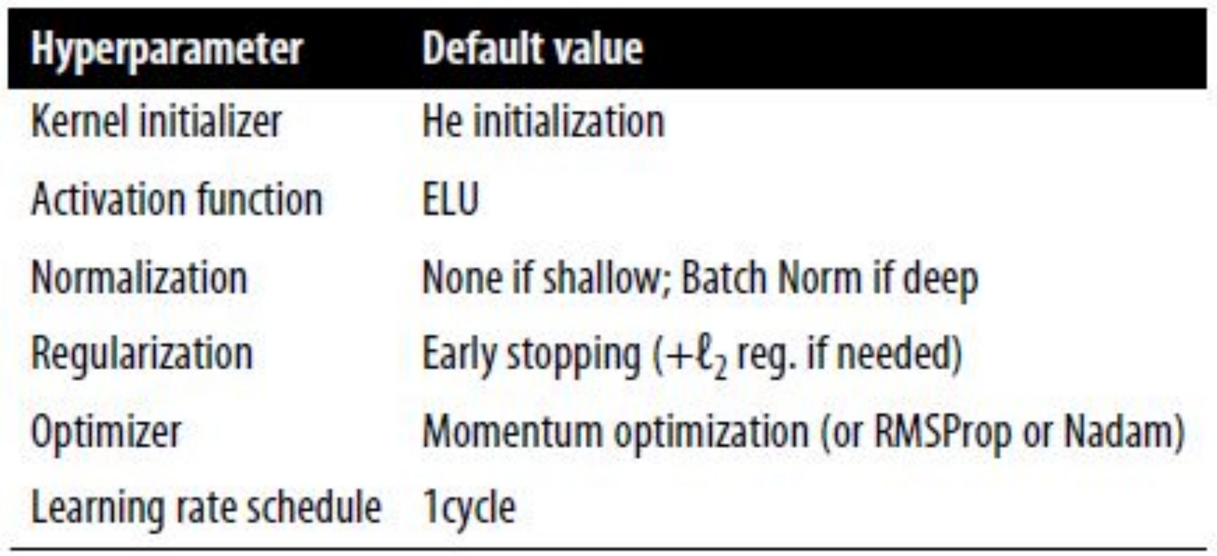
$\| \mathbf{w} \|_2 \leq r$, where $r$ is the max-norm hyperparameter and $\| \cdot \|_2$ is the $\ell_2$ norm.

Keras Implementation



```
keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal",
                   kernel_constraint=keras.constraints.max_norm(1.))
```

## Summary of Hacks to Develop ANN



| Hyperparameter | Default value |
|---|---|
| Kernel initializer | He initialization |
| Activation function | ELU |
| Normalization | None if shallow; Batch Norm if deep |
| Regularization | Early stopping (+$\ell_2$ reg. if needed) |
| Optimizer | Momentum optimization (or RMSProp or Nadam) |
| Learning rate schedule | 1cycle |