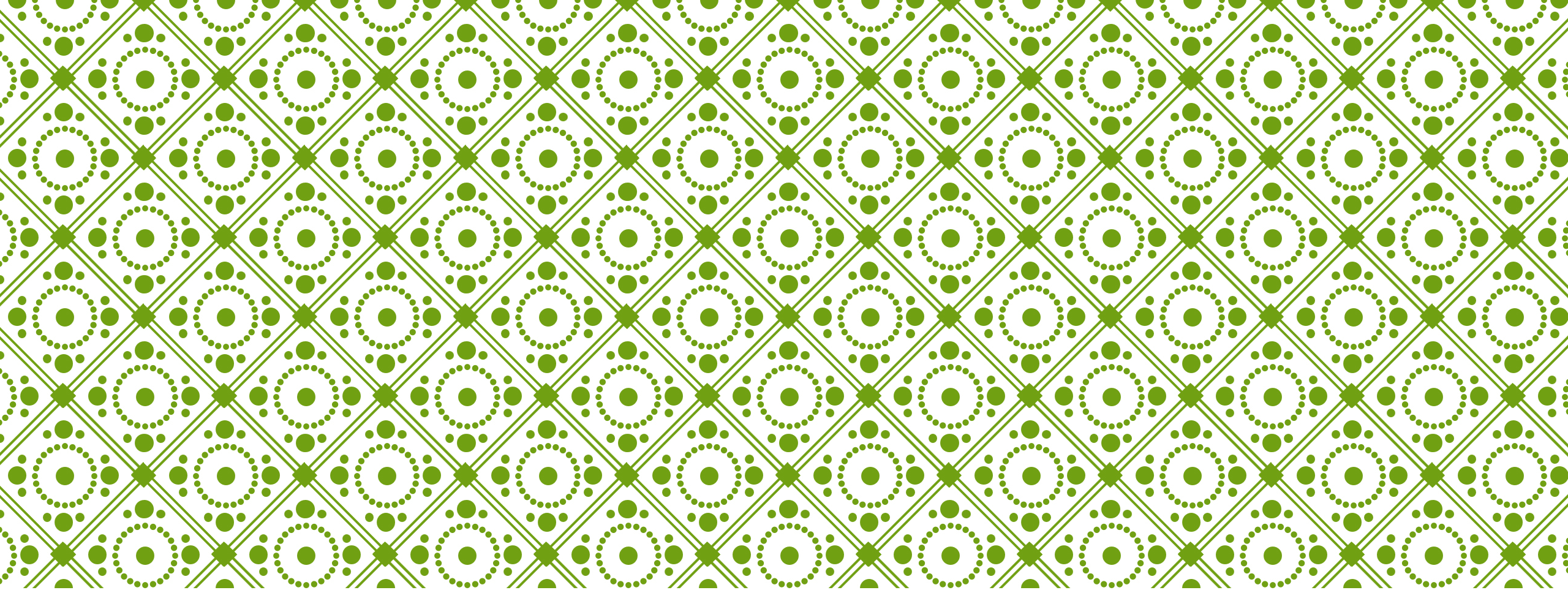


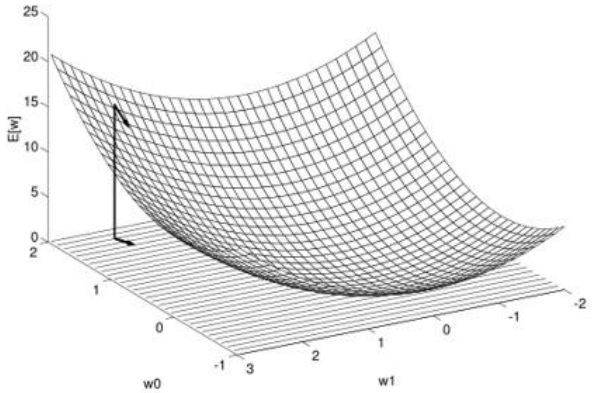
DEEP LEARNING BASICS

LIMITATIONS OF GRADIENT DESCENT

Gradient descent is an important general paradigm for learning

It is a strategy for searching through a large or infinite hypothesis space that can be applied whenever:

- The hypothesis space contains continuously parameterized hypothesis (e.g. weights in a linear unit)
- The error can be differentiated with respect to these hypothesis parameters



The key practical difficulties in applying gradient descent algorithm are:

- Converging to a local minimum can sometime be very slow (i.e. it can take many thousands of gradient descent steps)
- If there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum

STOCHASTIC GRADIENT DESCENT

- The **gradient descent** training rule computes **weight updates** after **summing over all the training examples in D**.
- The idea behind the stochastic gradient descent is to approximate this gradient descent search **by updating weights incrementally following the calculation of error for each individual example**
- One way to view the stochastic gradient descent algorithm is to consider a distinct error function $E_d(\mathbf{w})$ defined for each individual training example d
- By making the value of the learning rate η sufficiently small, stochastic gradient descent can be made to approximate the true gradient descent arbitrarily closely

Incremental (Stochastic) Gradient Descent

Batch mode Gradient Descent:
Do until satisfied

1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

Incremental mode Gradient Descent:
Do until satisfied

- For each training example d in D
  1. Compute the gradient $\nabla E_d[\vec{w}]$
  2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$E_d[\vec{w}] \equiv \frac{1}{2}(t_d - o_d)^2$$

Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η made small enough

- In standard gradient descent, the error is summed over all examples before updating weights
- In stochastic gradient descent, weights are updated upon examining each training example
- Summing over multiple examples in standard gradient descent requires more computation per weight update step.
- However, we can use a larger step size per weight update than stochastic gradient descent
- Multiple local minima – stochastic gradient descent can sometimes avoid falling into these local minima

Incremental (Stochastic) Gradient Descent

Batch mode Gradient Descent:
Do until satisfied

1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

Incremental mode Gradient Descent:
Do until satisfied

- For each training example d in D
  1. Compute the gradient $\nabla E_d[\vec{w}]$
  2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$E_d[\vec{w}] \equiv \frac{1}{2}(t_d - o_d)^2$$

Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η made small enough

CONVERGENCE AND LOCAL MINIMA [1]

- Gradient descent through the space of possible network weights → reduce the error between training example target values and network outputs
- Error surface for multilayer networks may contain many different local minima → trapping gradient descent
- Back Propagation over multilayer networks is guaranteed to converge towards some local minimum and not necessarily the global minimum term
- The problem of local minima has not been found to be as severe as feared. Why?

CONVERGENCE AND LOCAL MINIMA [2]

Impact of multiple weights

- A local minimum with respect to one weight will not necessarily be in a local minimum with respect to other weights
- More weights → more dimensions that provide "escape routes" for gradient descent to fall away from the local minimum with respect to this single weight

How network weights evolve with an increase in the training iterations

- Initialization of network weights close to zero → early gradient steps
  - Very smooth function that is approximately linear in its inputs
    - Sigmoid threshold function is approximately linear when the weights are close to zero
  - As the weights grow with increasing iterations, they represent highly non-linear network functions
- More local minima exist in the region of the weight space that represents these more complex functions (non-linear functions)
- By the time the weights reach this point, they have moved close to the global minima
  - Even local minima in this region are acceptable

CONVERGENCE AND LOCAL MINIMA [3]

Gradient descent over the complex error surfaces represented by ANNs is still poorly understood

- No methods that will predict with certainty when local minima will cause difficulties

Common Heuristics to Help Alleviate this Problem

- Add a momentum term to the weight update rule
  - Momentum term can sometimes carry gradient descent procedure through a narrow local minima

$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

- Use stochastic gradient descent (SGD) rather than true gradient descent
  - SGD effectively descends a different error surface for each training example
  - Different error surfaces typically will have different local minima → less likely to get stuck
- Train multiple networks with the same data but initializing them with different random weights → choose the network with best performance over separate validation data
- Committee of networks → output is weighted average of the individual network outputs

REPRESENTATIONAL POWER OF FEEDFORWARD NETWORKS[1]

Width and Depth of Networks determine the set of functions that can be represented by feedforward networks

Boolean Functions

- Every function can be represented exactly by some network with two layers of units
  - The number of hidden units required grows exponentially in the worst case with the number of inputs
- **Representing an arbitrary Boolean function**
  - For each possible input vector create a hidden unit
    - Set its weights so that it activates only if and only if this specific vector is input to the network
  - A hidden layer than will always have exactly one unit active
  - Implement an output unit as an OR gate that activates just for the desired input patterns

REPRESENTATIONAL POWER OF FEEDFORWARD NETWORKS[2]

Continuous Functions

- Every bounded continuous function can be approximated with arbitrarily small error (under a finite norm) by a network with two layers of units. (Cybenko, 1989)
- Networks use sigmoid units at the hidden layer
- (Unthresholded) linear units at the output layer
- The number of hidden units required depends on the functions to be approximated

REPRESENTATIONAL POWER OF FEEDFORWARD NETWORKS[3]

Arbitrary Functions

- Any function can be approximated to arbitrary accuracy by a network with three layers of units (Cybenko, 1988)
- Output layer → linear units
- Two hidden layers → sigmoid units
- Number of hidden units at each layer is not known in general
- Proof shows that any function can be approximated by a linear combination of many localized functions
  - Localized functions
    - 0 value everywhere except for some small region
  - Then, the two layers of sigmoid units are sufficient to produce good local approximations

HIDDEN LAYER REPRESENTATIONS[1]

One intriguing property of BP is its **ability to discover useful and intermediate representations at hidden unit layers** inside the network

Training examples constrain the inputs and outputs

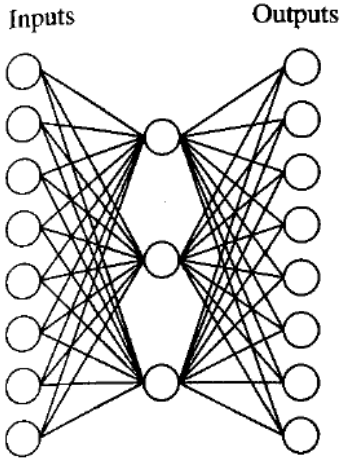
The weight tuning procedure is free to set the weights that define whatever hidden unit representation is most effective at minimizing the squared error E.

BP can define new hidden layer features that are:

- Not explicit in their input representation
- But can capture properties of the input instances that are most relevant to learning the target function.

HIDDEN LAYER REPRESENTATIONS[2]

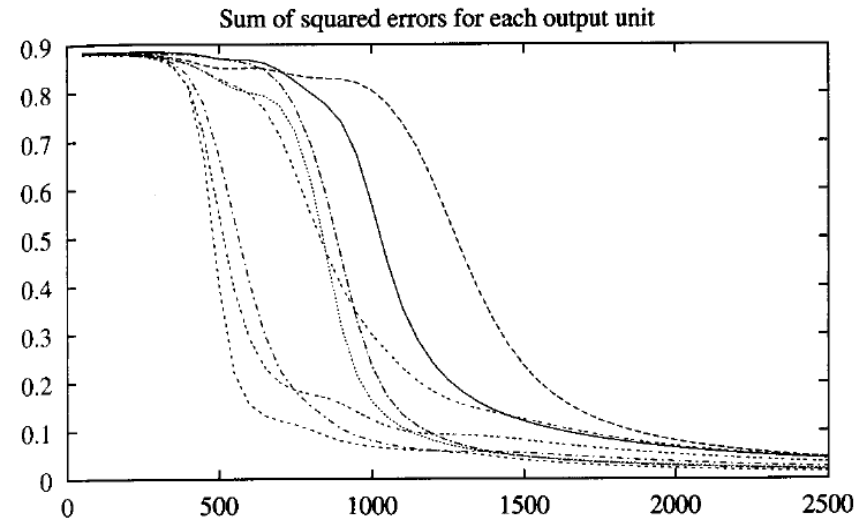
- The 8x3x8 network was trained to learn the identity function
- Eight training examples
- 5000 training epochs
- The three hidden units encode the eight distinct inputs using the encoding shown on the right
- If encoded values are rounded to zero or 1, the result is the standard binary encoding for eight distinct values
- Ability to automatically discover useful representations is a key feature of ANN learning
- Not constrained to use features defined by the human designer

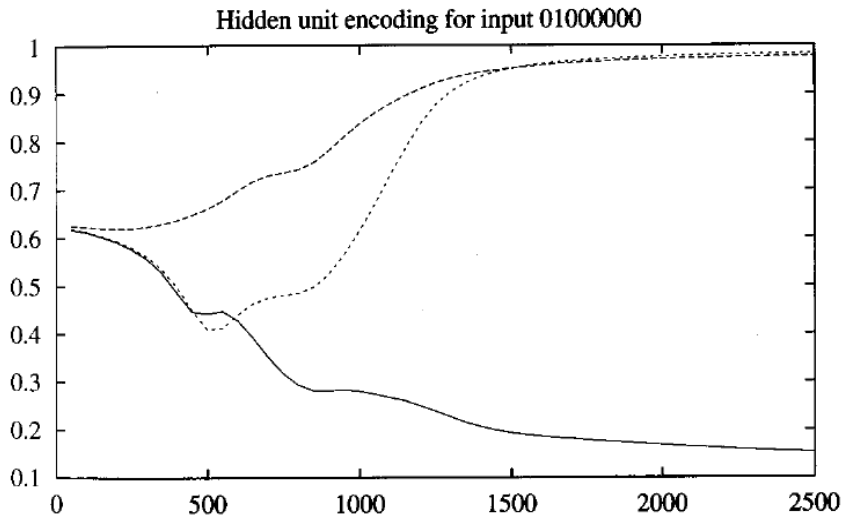


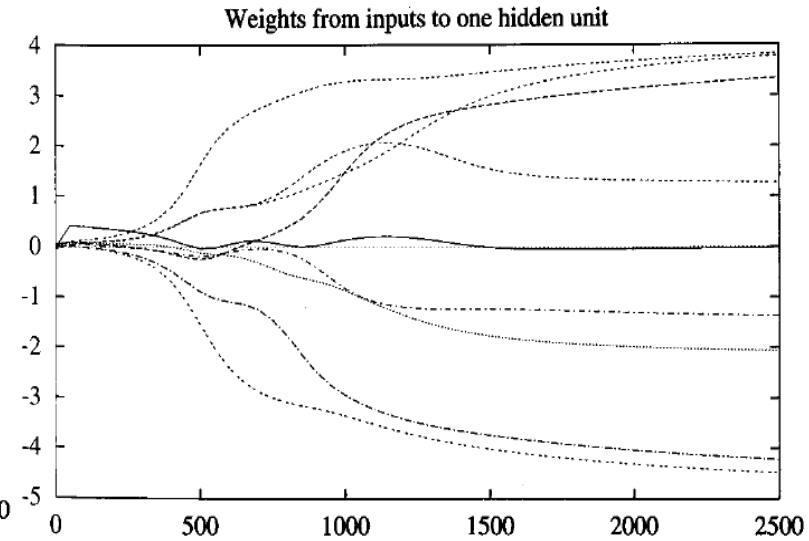
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

- The learner can invent new features
- When more layers of units are used in the network, more complex features can be invented

HIDDEN LAYER REPRESENTATIONS[3]







Squared output error plotted as a function of the number of gradient descent steps – each line depicts for one of the eight output units

For the hidden layer representation, the network passes through a number of different encodings before converging to the final encoding (as given in the previous slide)

The third graph represents the evolution of individual weights within the network

- Eight input units to one of the hidden units
- Significant changes here coincide with the hidden layer encoding and output layer errors significant changes
- The bias weight w_0 converges to a value near zero

IMPROVING BP BASED LEARNING

Over many years of development, various "tricks" have been proposed to improve the above basic learning protocol for the multilayer network:

- theoretically motivated
- inspired by pragmatisms

Activation functions

- Thresholding, identity, sigmoid, squared function and tanh functions
- Squared function and tanh have not proved to be useful due to practical learning and stability issues

ACTIVATION FUNCTIONS: RELU

The Rectified Linear Unit (ReLU) or rectifier is one of the most commonly used modern activation functions

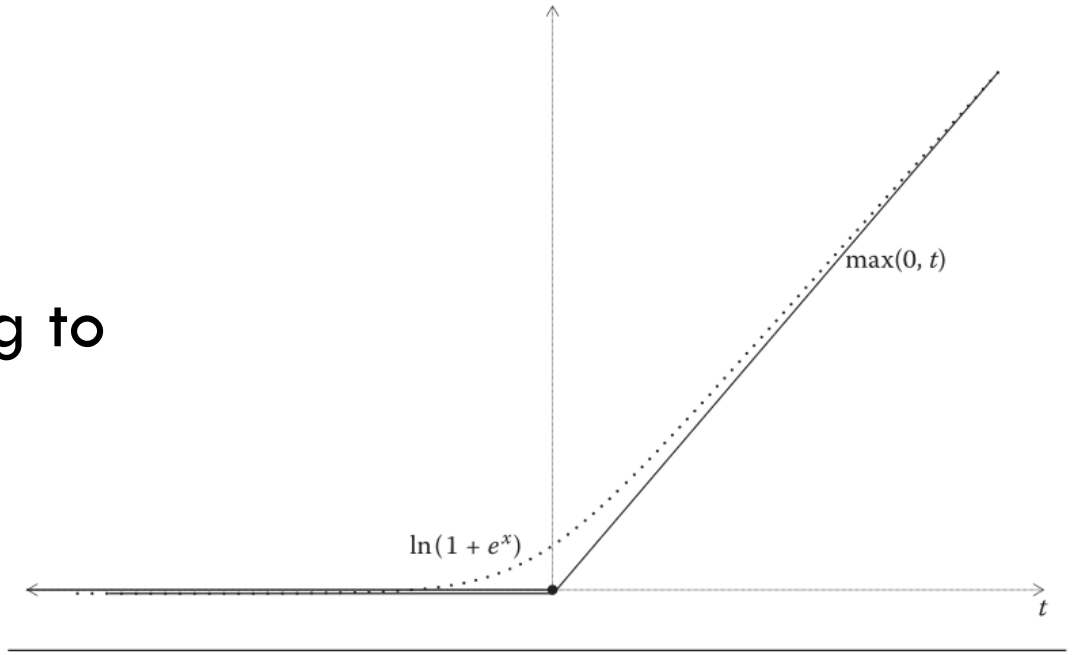
The rectifier is of the form

$a_{\mathrm{ReLU}}(t) = \max(0, t)$ which is not smooth

Actual implementation: $a'_{\mathrm{ReLU}}(t) = \ln(1 + e^x)$

Also called softplus.

ReLU has a much steeper profile leading to faster and better learnability

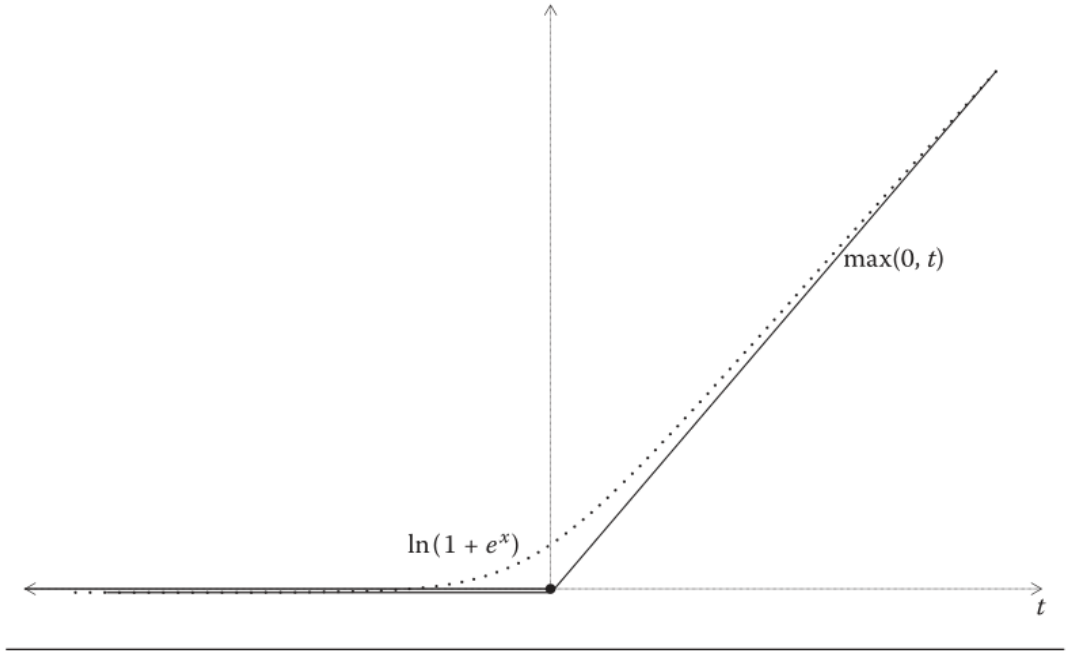


The ReLU activation function and its analytical approximation.

ACTIVATION FUNCTIONS: NOISY RELU

- Extension of Rectified Linear Unit (ReLU) called as noisy ReLU
- Noisy ReLUs create random errors in the activation that allow for the activations to be minutely wrong.
- This allows the network to wander off a little bit while learning.
- Consider the case where the error surface is full o peaks and valleys.
- Allowing the parameters to wander around in a restricted fashion helps in probing parts of the space that might not have been accessible using strict gradient descent
- Alleviate problems of overfitting
- Work well in some computer vision tasks

$$a_{\mathrm{ReLU}}(t) = \max(0, t + \varphi), \quad \text{with } \varphi \sim \mathcal{N}(0, \sigma(t))$$

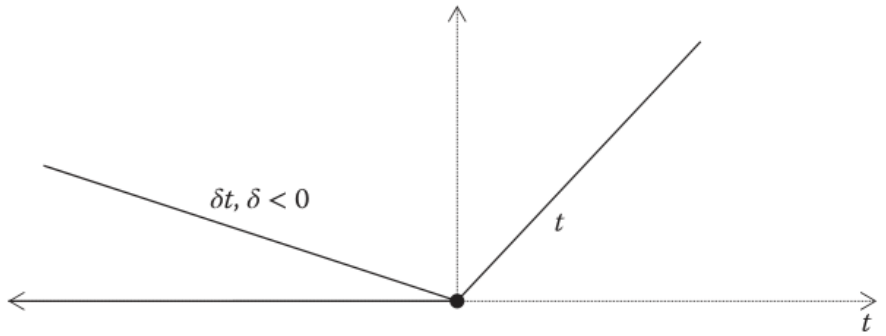


The ReLU activation function and its analytical approximation.

ACTIVATION FUNCTIONS: LEAKY RELUS

- The parameter δ is a small constant.
- This allows the neuron to be active very mildly and produces a small gradient no matter whether the neuron was intended to be active or not.
- A further development on this is the parametric leaky ReLU, where δ is considered another parameter of the neuron and is learned along with the BP of the weights themselves.
- If $\delta < 0$, an interesting activation function is obtained.
- The neuron quite literally produces a negative signal.
- This makes the gradients move much faster even when the neuron is not contributing to the class predictions.

$$a_{\mathrm{ReLU}}(t) = \begin{cases} t, & \text{if } t > 0 \\ \delta t, & \text{otherwise} \end{cases}$$



ACTIVATION FUNCTIONS: MAXOUT

- Recent activation function

$$a_{\text{maxout}}(t_i) = \max_{j \in [1,k]} t_{i,j}$$

- Quite general
- It can simulate any activation function from a linear rectifier to a quadratic.
- Maxout considers the neighboring k nodes' outputs
- It then produces the maximum of those outputs as the activation.
- Maxout **reduces the number of features that are produced by dropping those features that are not maximum enough.**
- We can consider that maxout assumes (forces) that nearby nodes represent similar concepts and picks only one of them, which is the most active
- Maxout's performance and value have not been fully understood – area of active research

WEIGHT PRUNING [1]

- A typical MLNN will have a large number of weights depending on the number of layers of a network and the number of nodes in each hidden layers
- Large number of weights result in a large degree of freedom : an overcomplicated system.
- Complicated systems suffer from two obvious disadvantages:
  - high computational complexity (and thus difficult to train)
  - tendency to overfitting (and thus poor generalization performance after training).
- Given a difficult learning task, it is also challenging to determine in advance the optimal size for a low-complexity network that still does an adequate job for the given task.

WEIGHT PRUNING [2]

- A common approach: start with an obviously larger-than-necessary network and then prune the network by deleting the weights and/or nodes to obtain a simpler network.
- The above weight-pruning task may be achieved by two types of approach or their variants.
  - The first type of approach employs some heuristics in selectively removing the weights if they are deemed as having little impact on the final error/cost function.
  - The second type relies on introducing additional regularization such as L1 terms in the error function so that smaller weights will be favored (essentially pushing some weights to zero).
- How to deal with the irregularity of a network that has gone through a weight- pruning process

BATCH NORMALIZATION [1]

- Images are normalized before feeding them forward through a network.
- Normalization involves ranging the image values to $[-0.5, 0.5]$, typically with a mean of 0.
- In a deep network, the input distribution of each layer keeps varying per batch and per sample.
- This is because the parameters of the previous layers change during every update.
- Makes training very difficult particularly with activation functions that saturate.
- The assumption that all samples from the same class are sampled independently and identically is not always true.
- Covariate shift: Phenomenon where samples differ in their statistical properties across batches of data even among the same class.
- To fix this problem of covariate shift, the activations coming off every layer are normalized

BATCH NORMALIZATION

- The right variance to normalize with and the mean (subtract the data) are often unknown and can be estimated from the dataset itself.

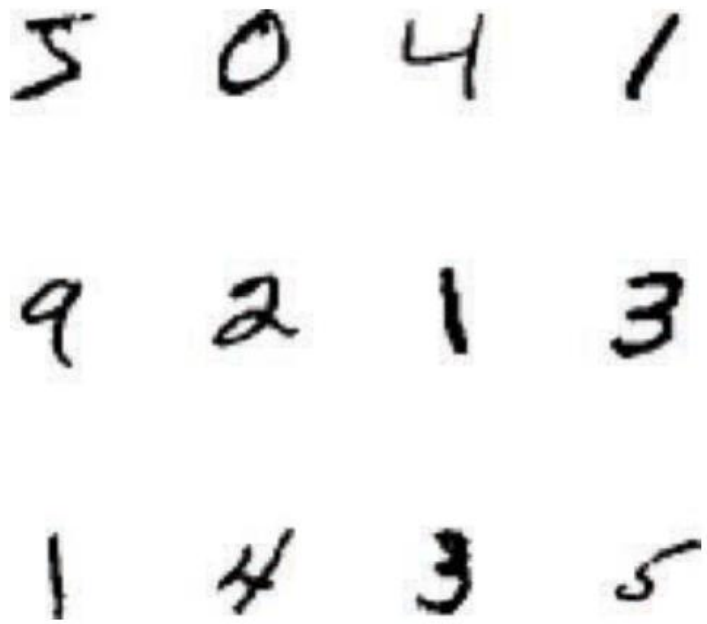
- If z were the activations of one layer, we compute $$z_{bn} = \frac{(z - \mu_z) * \alpha_z}{\sigma_z}$$

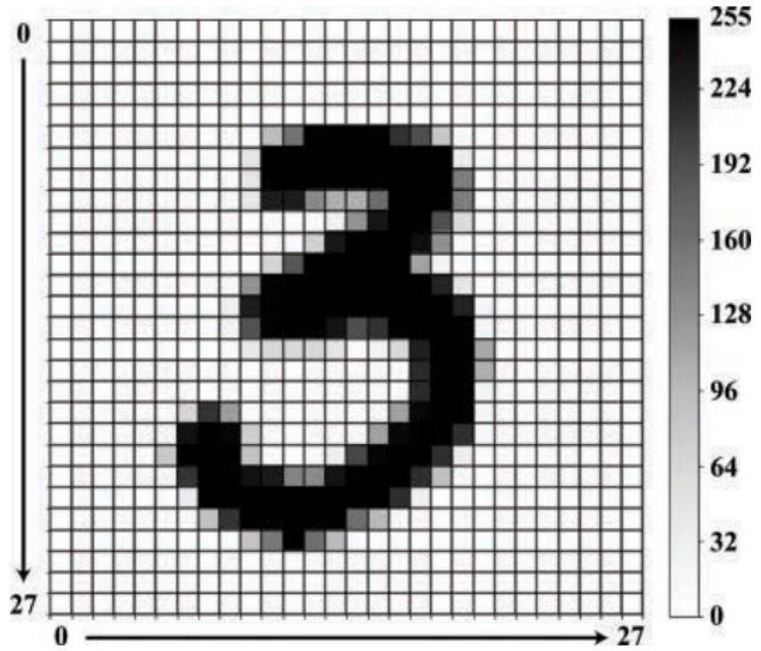
- where μ_z and σ_z are the mean and the variance of that activation batch, respectively.
- α is now one of the learnable parameters of the network and can be thought of as learning the stretch of the normalization applied.
- α is learned during the same optimization along with the weights.
- α can be learned for multiple layers using BP.
- Batch normalization is a powerful tool and helps the network to learn much faster even with non-saturating activation functions.
- Batch normalization is particularly popular in visual computing contexts with image data.

SHALLOW NET IN KERAS

SHALLOW NET IN KERAS

- The MNIST Handwritten Digits
- One of the more popular datasets for Deep Learning Tutorials
- Each image contains a single digit handwritten by either a high school student or a U.S. census worker
- Small by modern standards
  - 60,000 images for training
  - 10,000 images for validation
  - Rapid modeling possible
  - 28 x 28 pixel image
  - Each pixel is 8-bit
  - Pixel darkness: 0 (white) to 255 (black)
- Samples are sufficiently diverse and contain complex enough details
- Not easy for an ML algorithm to identify with high accuracy
- Not insurmountable





SCHEMATIC DIAGRAM OF THE NETWORK [1]

Total of three layers with one hidden layer

Shallow network

First layer: input

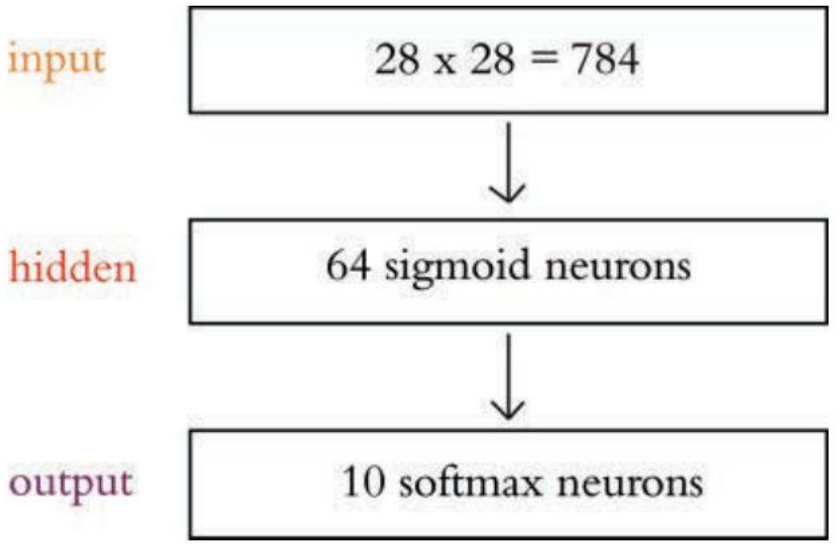
- 28 x 28 pixel images
- 784 values
- Flatten the two-dimensional 28 x 28 shape to a one-dimensional array of 784 elements
- Loss of information when we move from 2 dimensions to 1 dimension

Second Layer

- Hidden layer of 64 sigmoid neurons
- Responsible for learning representations of the input data

Output Layer

- 10 Softmax neurons



SCHEMATIC DIAGRAM OF THE NETWORK [2]

Output Layer

- 10 Softmax neurons

Sigmoid neuron → binary classification

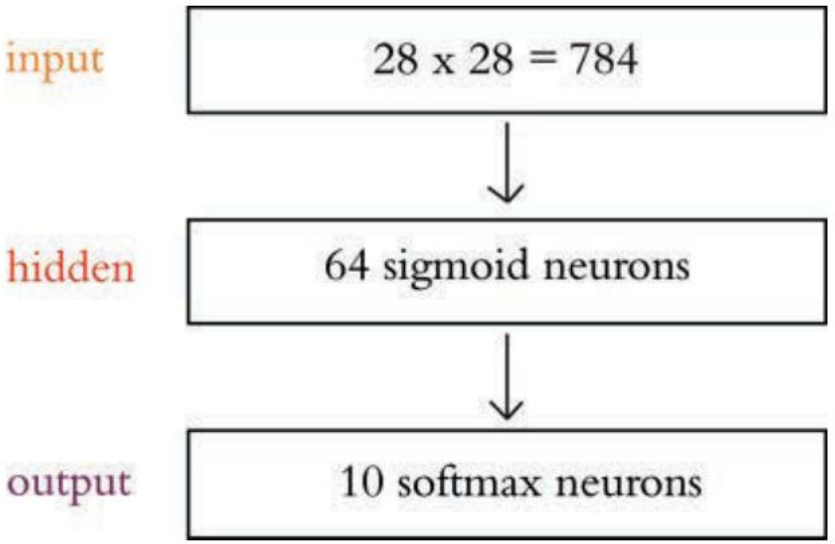
A softmax neuron layer is used for multi-class classification

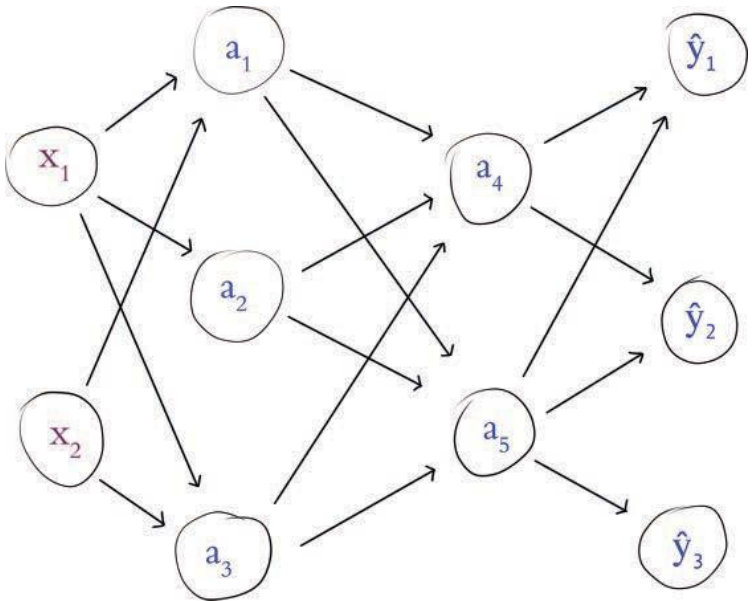
- 10 classes – 0, 1, .. , 9

The softmax function involves three steps

- Calculate the exponential of each of the inputs to the nodes
- Sum up the exponentials for all the nodes in the output layer
- Calculate the proportions for each of the class, relative to all the classes

The function returns the class with the highest probability – instead of a [0, 1]



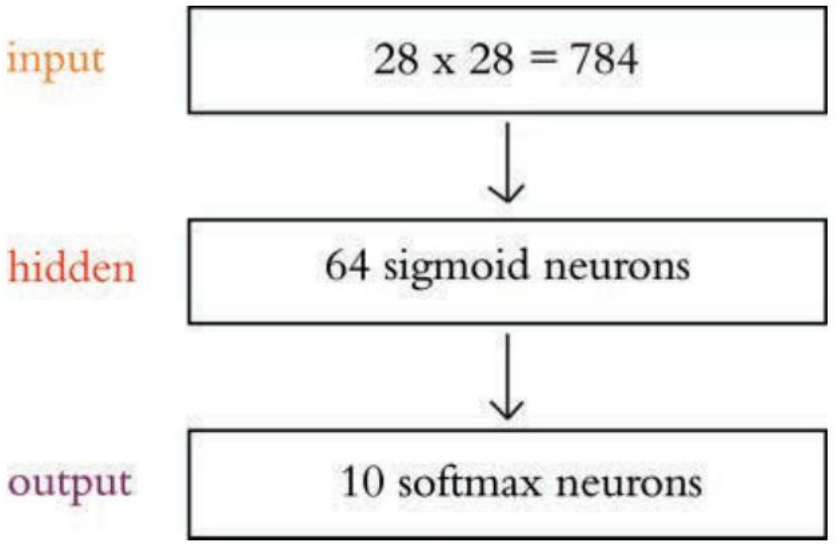


SHALLOW NET IN KERAS

- Loading the data
- Import the software dependencies
- Load the MNIST data
  - Inspect sample of the data
  - Use matplotlib to show the picture of the digits
- Rearrange the input data to match the shapes of the input and output layers of the network
  - Flatten the two-dimensional images to one dimension
  - Convert pixel integers to floats
  - Convert integer labels to one-hot format
    - 10 classes
    - Utility function to convert training and validation labels from integers to one-hot format
    - The labels line up with the 10 probabilities being output by the final layer of our ANN.

DESIGNING A NEURAL NETWORK ARCHITECTURE

```
#Keras Code to architect a shallow network
model = Sequential()
model.add(Dense(64, activation='sigmoid', input_shape=(784,)))
model.add(Dense(10, activation='softmax')) # Sepecifies the out
```



Instantiate the simplest type of neural network object model: **the Sequential type**

- Each layer in the network passes information only to th next layer in the **sequence** of layers
- **Dense** – fully connected arrangement
- 10 artificial neurons of the softmax variety that corresponds to the 10 probabilities

TRAINING THE DEEP LEARNING MODEL

```
model.fit(X_train, y_train, batch_size=128,
          epochs=200, verbose=1, validation_data=(X_valid, y_valid))
```

Fit method is used to train our ANN with the training images X_train

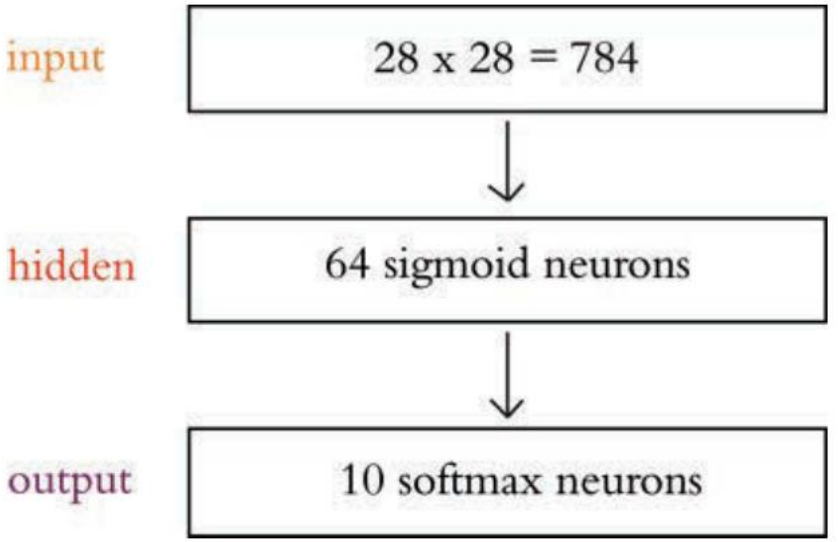
Fit() method also provides an option to evaluate the performance of the network by passing the validation data

In one epoch, the deep neural network cycles through the 60,000 images in batch sizes of 128

- 200 epochs

Validation Accuracy: Validation accuracy is the proportion of the 10,000 handwritten images in X_valid in which the network's highest probability in the output layer corresponds to the correct digit as per the labels in y_valid.

The shallow network reaches 86% validation accuracy



REVISITING THE ACTIVATION FUNCTIONS [1]

Activation functions are threshold functions that output when the input crosses a certain threshold value

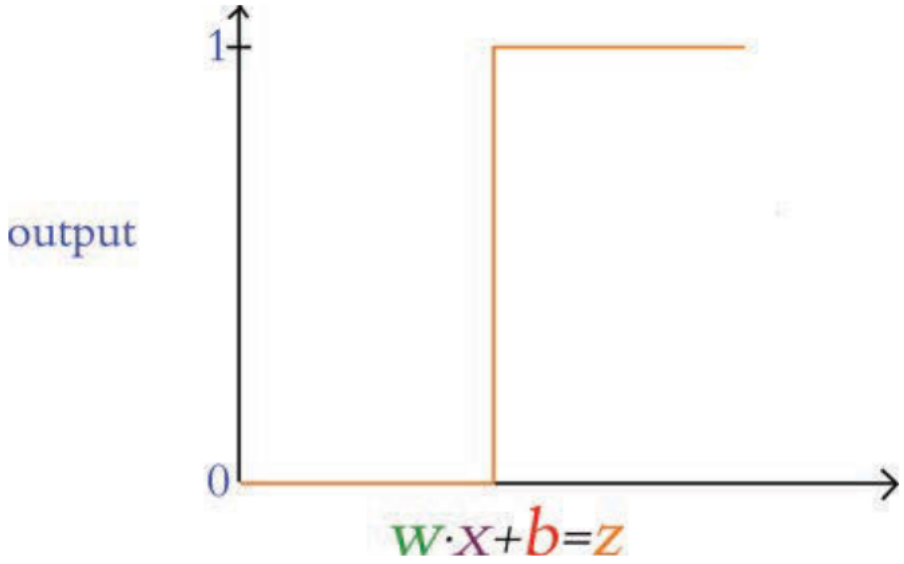
Perceptron

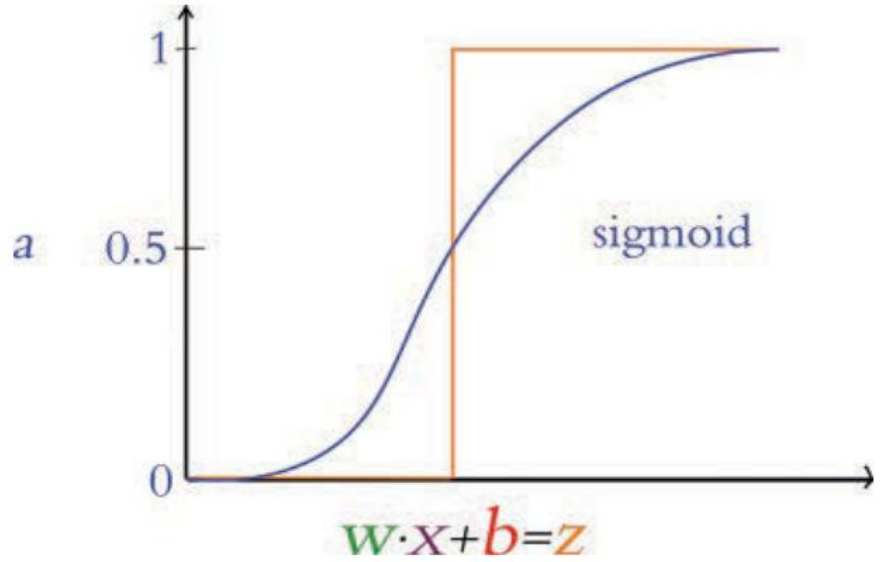
- Sudden transition from 0 to 1
- Not ideal for training

Sigmoid Function

- Gradual transition with small updates
- Saturation at very high or low values

$$\sigma(z) = \frac{1}{1+e^{-z}}$$



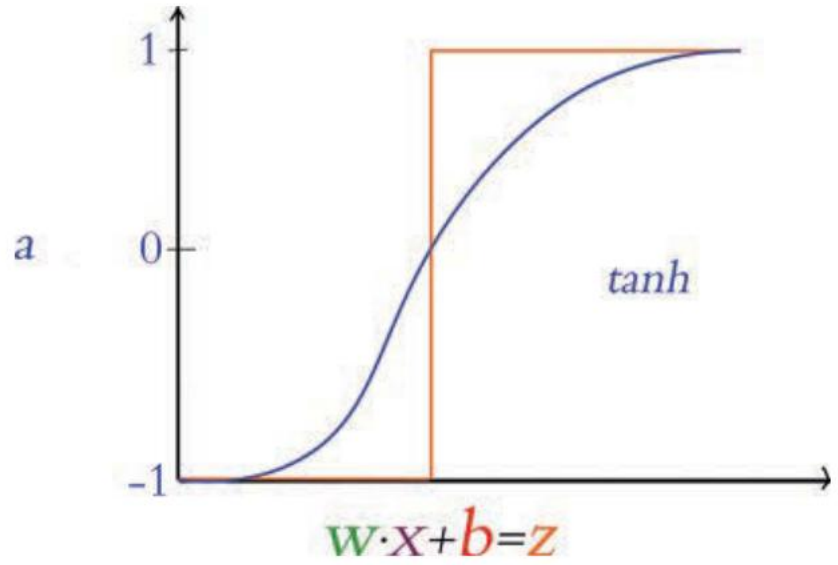


REVISITING THE ACTIVATION FUNCTIONS [2]

$$\sigma(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

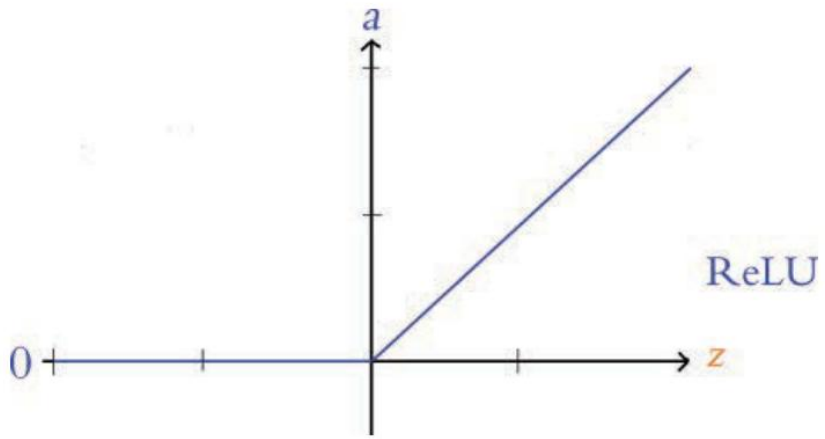
The Tanh Neuron

- Similar to the Sigmoid Function
- Sigmoid function range [0,1]
- Tanh function range [-1, 1]
  - Negative inputs – negative activations
  - Zero input – zero activation
- Zero centered inputs make it less likely to saturate



ReLU: Rectified Linear Units

- a = max(0, z)
- One of the simplest non-linear functions
- Permits it to approximate any continuous function
- More easy and efficient to train using the back propagation algorithm as compared to other activation functions



CHOOSING A NEURON

The perceptron, with its binary inputs and the aggressive step of its binary output, is not a practical consideration for deep learning models.

- Should be avoided

The **sigmoid** neuron is an acceptable option,

- Tends to lead to neural networks that **train less rapidly** than those composed of, say, tanh or ReLU neurons.
- Recommended for situations where it would be helpful to have a neuron provide output within the range of [0, 1].

The tanh neuron is a good solid choice

- The 0-centered output helps deep learning networks learn rapidly
- Prevents saturation

ReLU is the preferred neuron

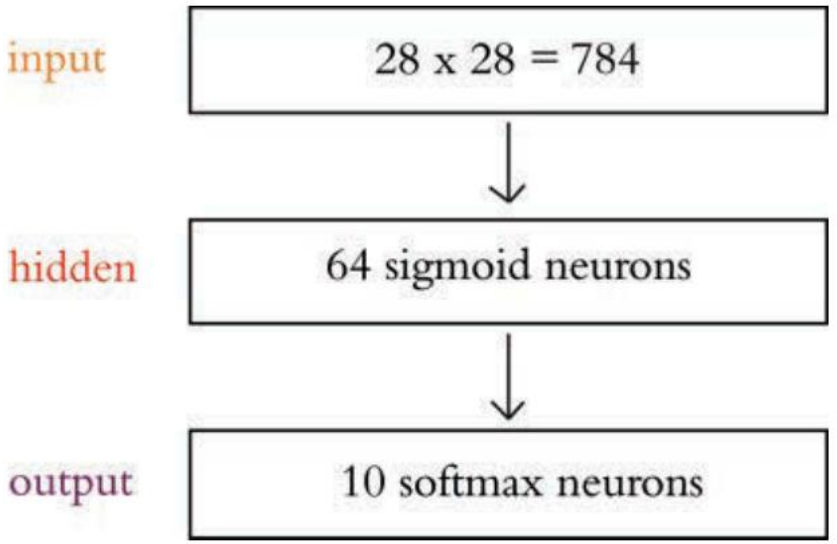
- Efficiency in enabling learning algorithms perform computations.
- Practically, they result in well-calibrated artificial neural networks in the shortest period of training time.

Other options include: leaky ReLU, parametric ReLU, the exponential linear unit, etc.

SHALLOW NETWORK IN KERAS

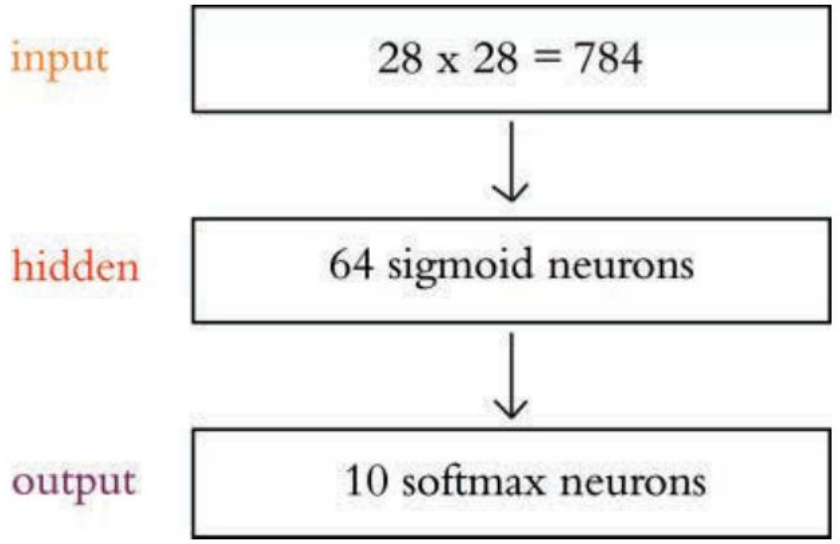
The Input Layer

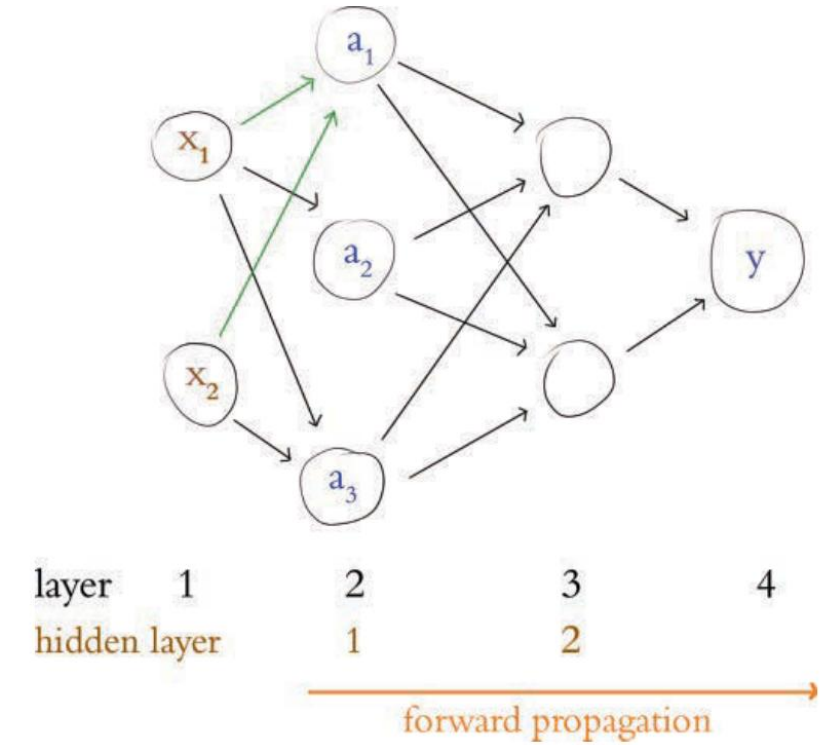
- Neurons in the input layer do not perform any calculations
- Placeholders for input data
- ANNs perform computations on matrices that have predefined dimensions
- One of these predefined dimensions in the network architecture corresponds directly to the shape of the input data



DENSE LAYERS

- Many kinds of hidden layers
- Dense layer is the most general type
- Can be called a fully connected layer
- Each of the neurons in a given dense layer receive information from every one of the neurons in the preceding layer of the network.
- In other words, a dense layer is fully connected to the layer before it.
- https://playground.tensorflow.org/





REVISITING THE SHALLOW NETWORK

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 64)                50240

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0
_________________________________________________________________
```

Has 50,240 parameters associated with it, broken down into the following:

- 50,176 weights, corresponding to each of the 64 neurons in this dense layer
- receiving input from each of the 784 neurons in the input layer (64×784)
- Plus 64 biases, one for each of the neurons in the layer
- Giving us a total of 50,240 parameters:

$$n_{parameters} = n_w + n_b = 50176 + 64 = 50240$$

Non-trainable parameters

COST FUNCTIONS

Help to quantify the spectrum of output-evaluations

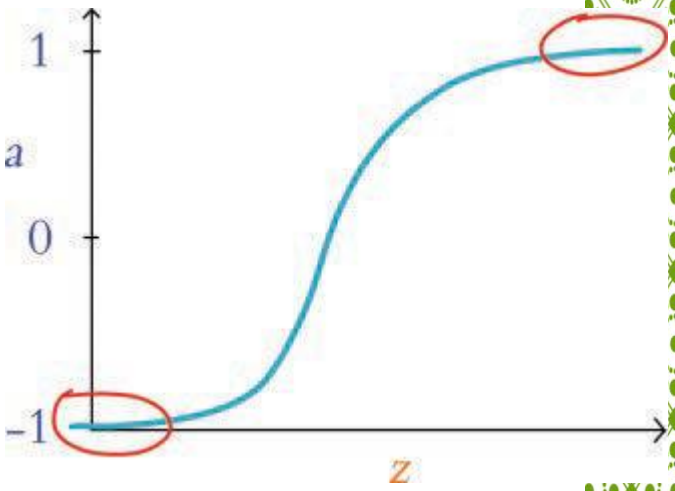
Quadratic Cost

- Mean square error cost

$$C = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Phenomenon of Saturated Neurons

- Learning slows down
- In a saturated neuron, where changes to ***w*** and ***b*** lead to only minuscule changes in a, this learning slows to a crawl
- Affect downstream learning
  - If adjustments to ***w*** and ***b*** make no discernible impact on a given neuron's activation a, then these adjustments cannot have any discernible impact downstream (via forward propagation) on the network's ***ŷ***, its estimate of ***y***.



COST FUNCTIONS: CROSS ENTROPY COST

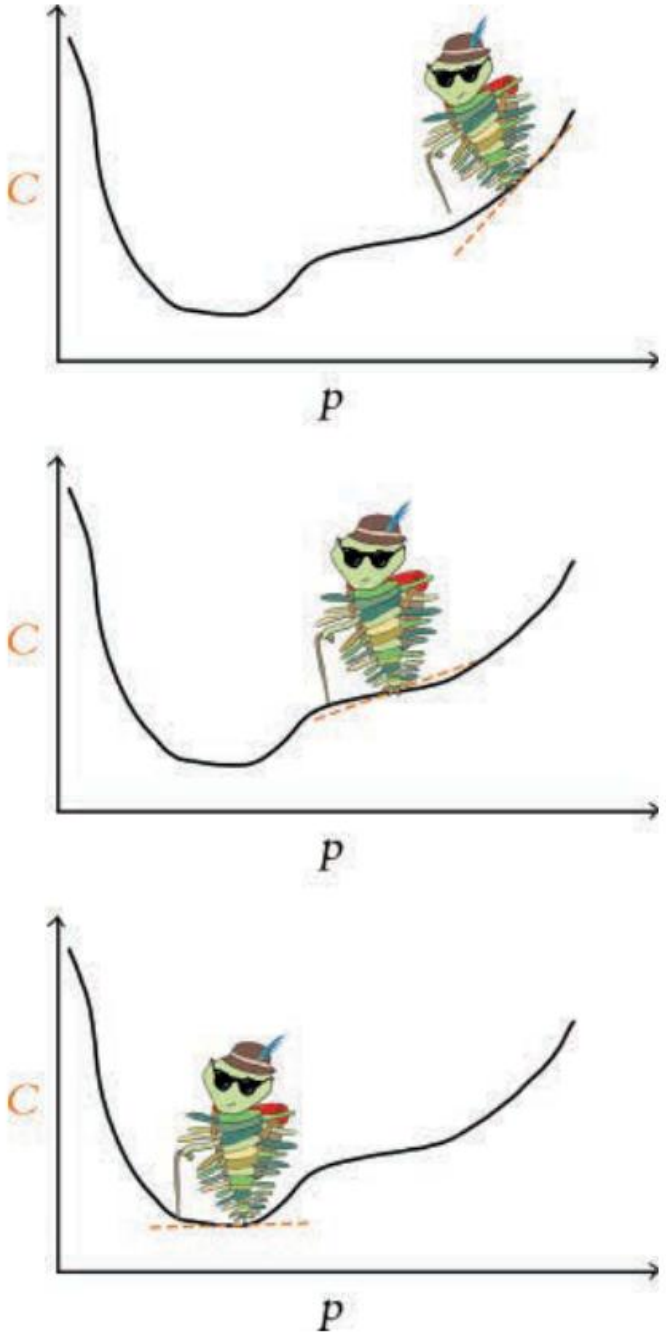
$$C = -\frac{1}{n}\sum_{i=1}^{n}[y_i \ln \hat{y}_i + (1 - y_i)\ln(1 - \hat{y}_i)] \qquad C = -\frac{1}{n}\sum_{i=1}^{n}[y_i \ln a_i + (1 - y_i)\ln(1 - a_i)]$$

- Like quadratic cost, divergence of ***ˆy*** from ***y*** corresponds to increased cost.
- Analogous to the use of the square in quadratic cost, the use of the natural logarithm ***ln*** in cross-entropy cost causes larger differences between ***ˆy*** and ***y*** to be associated with exponentially larger cost.
- Cross-entropy cost is structured so that the larger the difference between ***ˆy*** and ***y***, the faster the neuron is able to learn.
- The chief distinction between the quadratic and cross-entropy functions is not the particular cost value that they calculate per se, but
  - rather it is the rate at which they learn within a neural net
  - especially if saturated neurons are involved.

REVISITING GRADIENT DESCENT

Moves in the direction where the gradient is the maximum

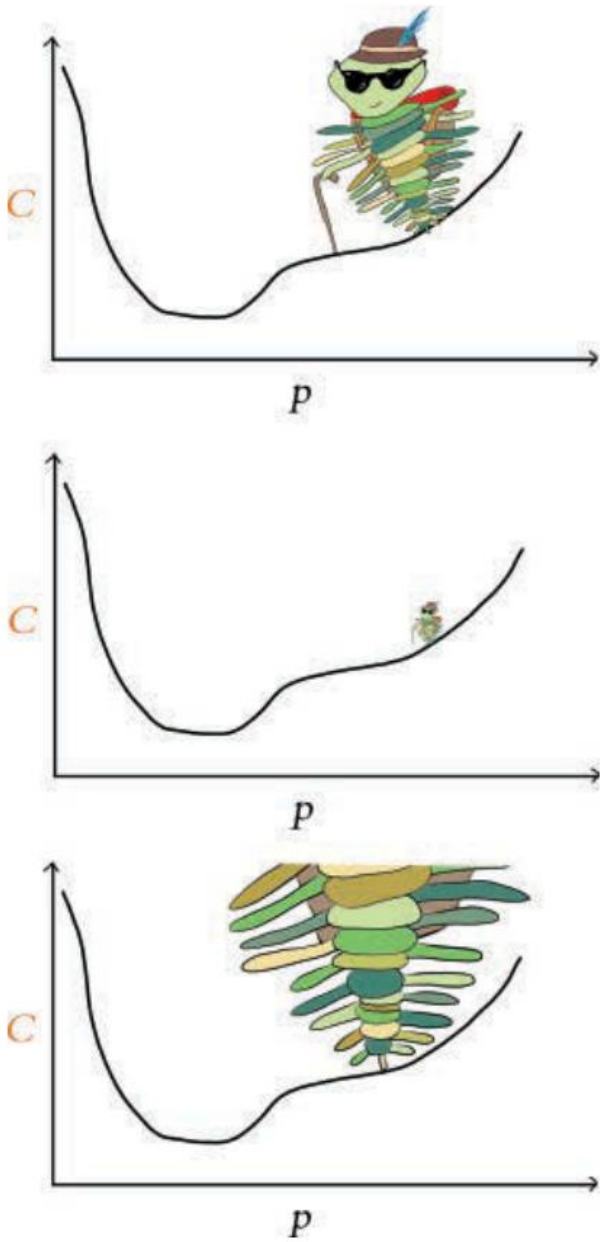
Higher Cost → Lower Cost



LEARNING RATE

The steps correspond to the learning rate – one of the hyperparameters

- Very small steps will result in a long training time – many iterations
- Very large steps might result in the global minima being missed
  - Jumps right over the parameters associated with the minimal cost
- Trial and error



LEARNING RATE – RULES OF THUMB

- Begin with a learning rate of about 0.01 or 0.001.
- Increase your learning rate by an order of magnitude (e.g., from 0.01to 0.1)
  - If the model is able to learn (i.e., if cost decreases consistently epoch over epoch)
  - but training happens very slowly (i.e., each epoch, the cost decreases only a small amount),
- If the cost begins to oscillate i.e. jumps up and down erratically epoch over epoch, then the learning rate is too high and it needs to be decreased
- At the other extreme, if the model is unable to learn,
  - then your learning rate may be too high.
  - Try decreasing it by orders of magnitude (e.g., from 0.001 to 0.0001) until cost decreases consistently epoch over epoch.

BATCH SIZE AND STOCHASTIC DESCENT

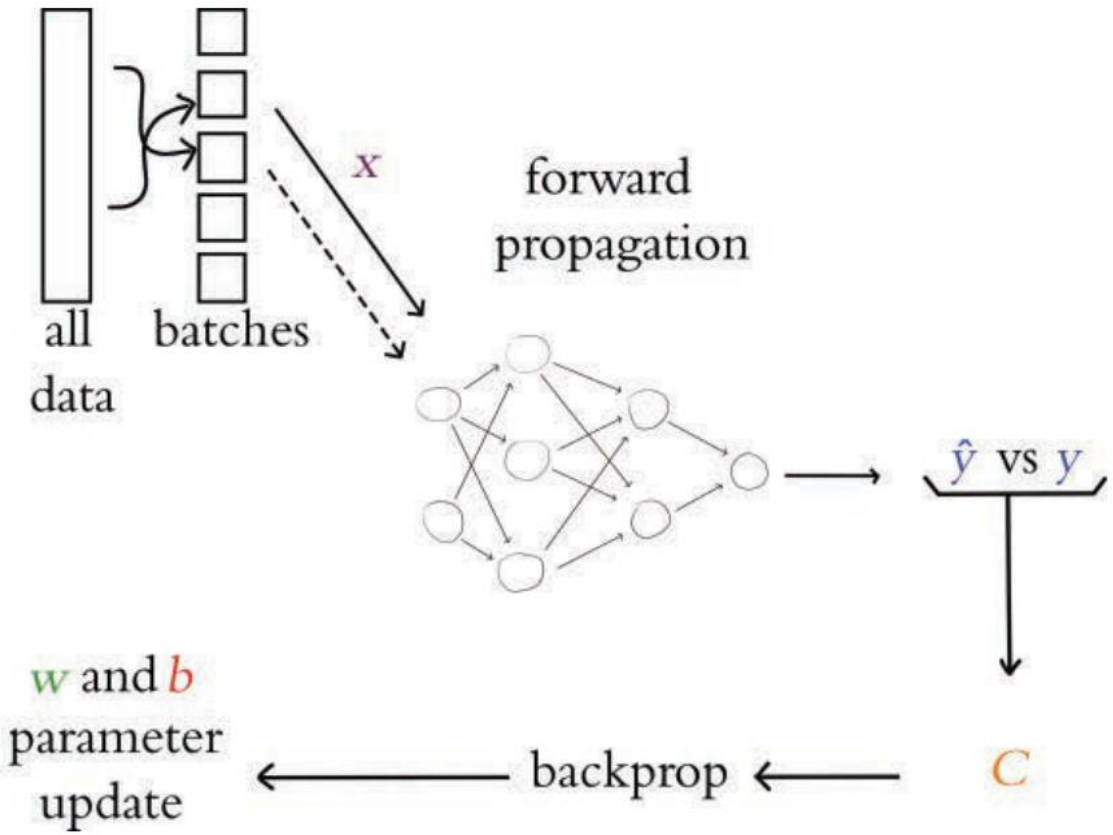
- For very large quantity of training data, ordinary gradient descent would not work at all
  - because it wouldn't be possible to fit all of the data into the memory (RAM) of our machine.
  - highly inefficient because of the computational complexity of the associated high-volume, high-dimensional calculations
- In stochastic gradient descent, the training data is split into mini-batches—small subsets of the full training dataset—to render gradient descent both manageable and productive.
- Batch size is another hyperparameter set by the user
- If Batch size = 128, for 60,000 images in the dataset, we have 60000/128 = 489 (approx.) batches of gradient descent per epoch.

```
model.fit(X_train, y_train, batch_size=128,
          epochs=200, verbose=1, validation_data=(X_valid, y_valid))
```

BATCH SIZE AND STOCHASTIC DESCENT

Round of Training:

1. Sample a mini-batch of x values
2. Forward propagate x through network to estimate y with $\hat{y}$
3. Calculate cost C by comparing y and $\hat{y}$
4. Descend gradient of C to adjust w and b, enabling x to better predict y



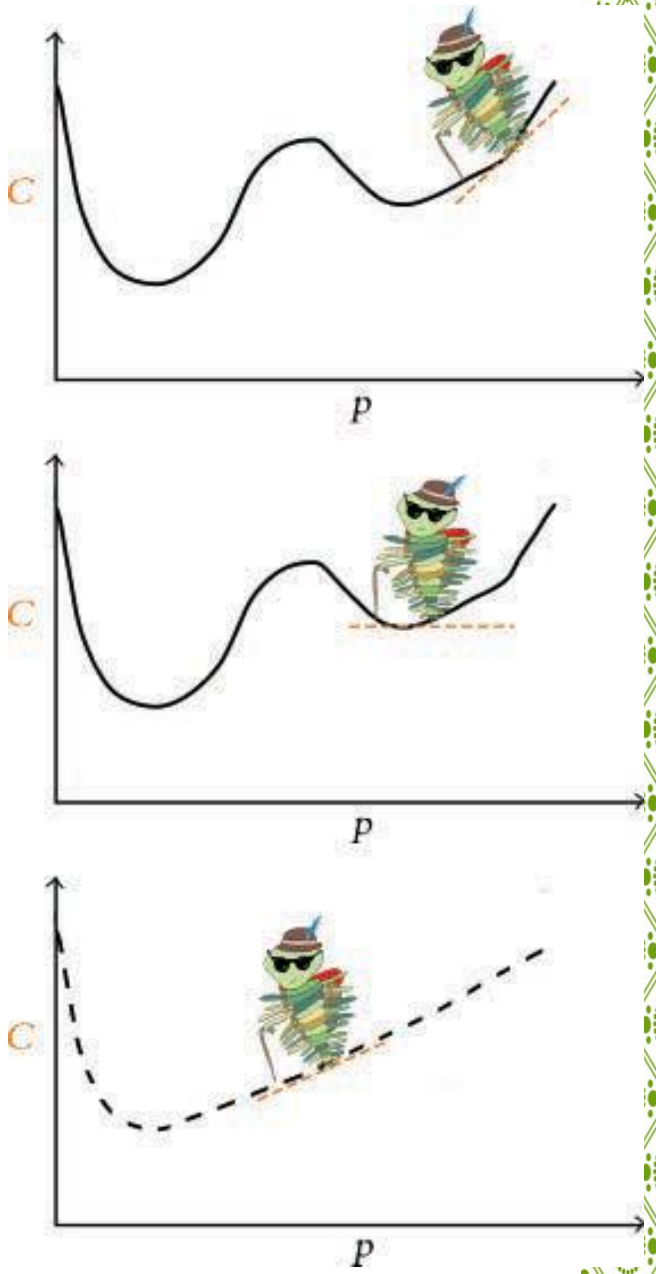
- The sampling is done without replacement
- At the end of an epoch, each image is seen by the algorithm only once
- Between epochs, the batches are sampled randomly

SETTING THE NUMBER OF EPOCHS

- The total number of epochs is one of the easiest hyperparameters to get right.
- If the cost on the validation data is going down epoch over epoch, and if the final epoch attained the lowest cost yet, then we can try training for additional epochs.
- Once the **cost on your validation data begins to creep upward**, that's an indicator that your **model has begun to overfit** to your training data because you've **trained for too many epochs**.
- There are methods that can be used to automatically monitor training and validation cost and stop training early if cost starts to rise.
  - The number of epochs could be set to be arbitrarily large
  - Training will continue until the validation cost stops improving
  - Before the model begins overfitting

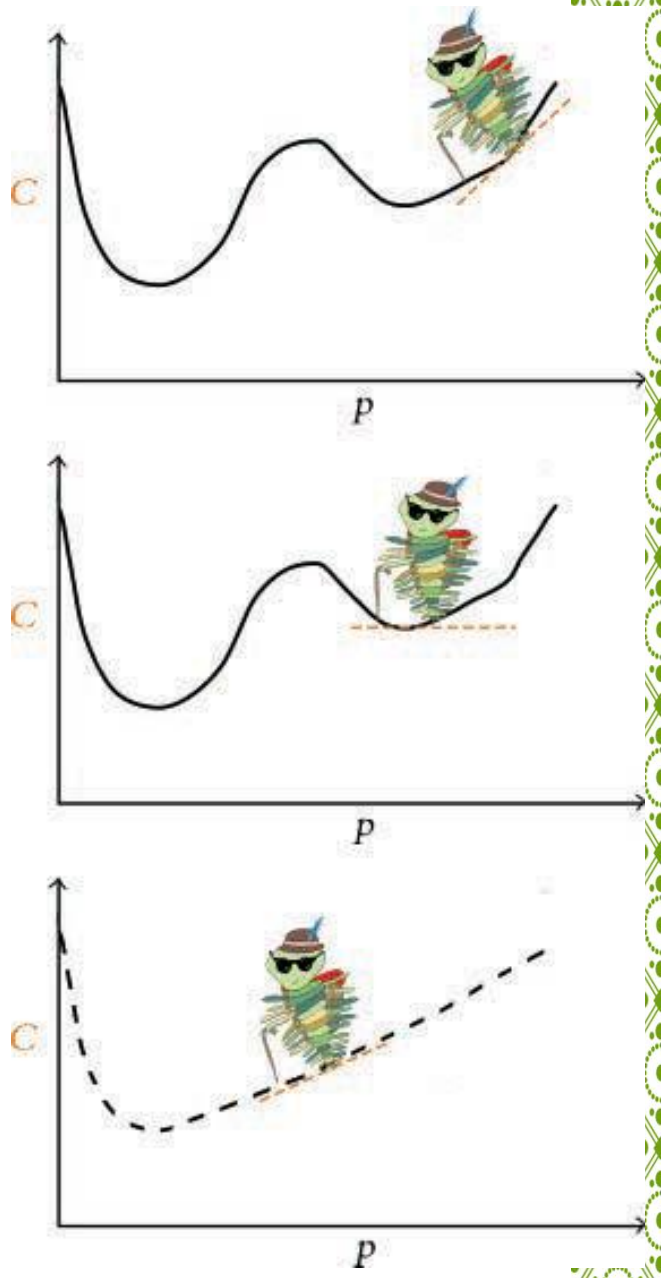
ESCAPING THE LOCAL MINIMUM[1]

- Complex relationship between the parameter ***p*** and cost **C**.
- The neural network estimates ***y*** accurately only if gradient descent is able to identify the parameter values associated with the lowest-attainable cost.
- However due to its random starting point, the gradient descent can get trapped in a local minimum.
- The sampling of mini-batches can have the effect of smoothing out the cost curve – as shown by the dashed curve shown in the bottom panel
- Smoothing happens because the estimate is noisier when estimating gradient from a smaller mini-batch (as opposed to the entire dataset)
- **Inaccurate estimate of the gradient enables the SGD algorithm to move forward – and avoid local minima**



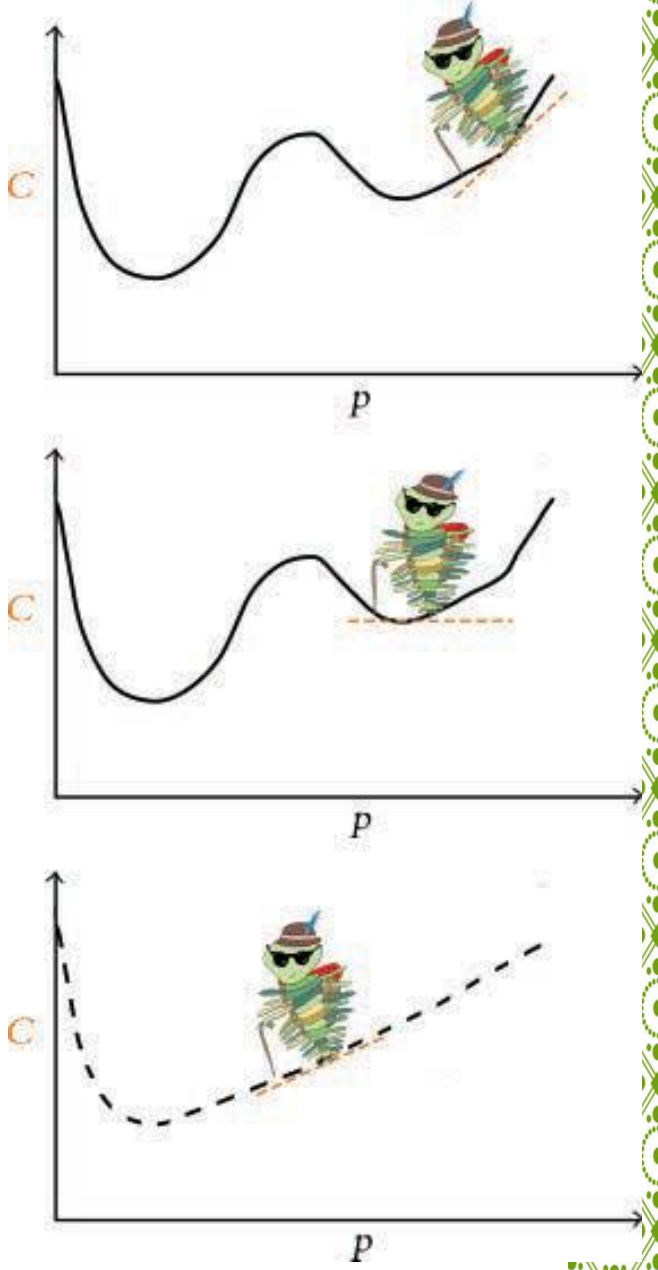
ESCAPING THE LOCAL MINIMUM[2]

- If the batch size is too large
  - Estimate of the gradient of the cost function is more accurate.
  - However, the model is at risk of becoming trapped in local minima
- If the batch size is too small
  - Each gradient estimate may be excessively noisy (because a very small subset of the data is being used to estimate the gradient of the entire dataset)
  - Training will take longer because of erratic gradient descent steps



ESCAPING THE LOCAL MINIMUM[3]

- Some rules of thumb
  - Start with a batch size of 32.
  - If the mini-batch is too large to fit into memory, the batch size is decreased by powers of 2 (e.g., from 32 to 16).
  - If the model trains well (i.e., cost is going down consistently) but each epoch is taking very long and enough RAM is available then a bigger batch size can be tried.
  - To avoid getting trapped in local minima, a batch size beyond 128 is not recommended



TUNING HIDDEN-LAYER COUNT AND NEURON COUNT[1]

- The **number of hidden layers** in the neural network is another hyperparameter.
- **Advantage:** Hidden layers help the neural network represent more abstract concepts
- **Disadvantage** of adding layers is that backpropagation becomes less effective:
  - Backprop has its **greatest impact** on the parameters of the hidden layer of neurons **closest to the output ^y**.
  - The farther a layer is from ^y, the **more diluted the effect** of that layer's parameters on the overall cost.

TUNING HIDDEN-LAYER COUNT AND NEURON COUNT[2]

Rules of thumb

- Start with about two to four hidden layers
- If reducing the number of layers does not increase the cost you can achieve on your validation dataset, the layers can be decreased:
  - **Occam's razor:** the simplest network architecture that can provide the desired result is the best
  - Train more quickly and require fewer compute resources.
- If increasing the number of layers decreases the validation cost, then number of layers can be increased

TUNING HIDDEN-LAYER COUNT AND NEURON COUNT[3]

- The **number of neurons/nodes in each hidden layer** in the neural network is another hyperparameter.
- Too many neurons: Additional Complexity
- Too few neurons: reduces network accuracy
- A sense for how many neurons might be appropriate in a given layer is developed by training more and more deep neural networks
- If there are lots of low-level features to represent then more neurons in the network's early layers are recommended
- If there are lots of higher-level features to represent, then having additional neurons in its later layers is beneficial

TUNING HIDDEN-LAYER COUNT AND NEURON COUNT[4]

Empirical experimentation:

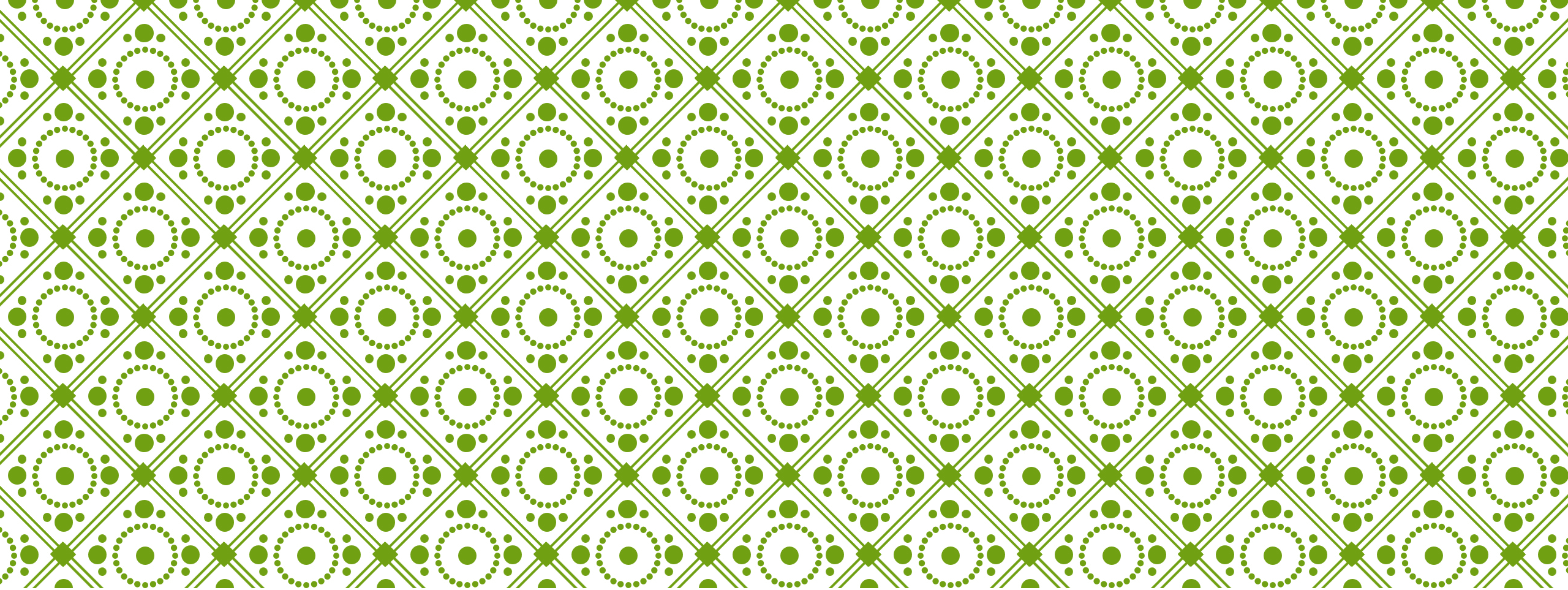
- Experiment with the neuron count in a given layer by varying it by powers of 2.
- Doubling the number of neurons (64 to 128) can be considered if it provides an appreciable improvement in model accuracy
- Occam's razor: Reduce the number of neurons (64 to 32) if it doesn't reduce the model accuracy
  - Model's computational complexity reduced with no apparent negative effects

AN INTERMEDIATE DEPTH NEURAL NETWORK

```
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(784,)))
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

```
model.compile(loss='categorical_crossentropy',
              optimizer=SGD(lr=0.1),
              metrics=['accuracy'])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 64)                50240
_________________________________________________________________
dense_2 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_3 (Dense)              (None, 10)                650
=================================================================
Total params: 55,050
Trainable params: 55,050
Non-trainable params: 0
_________________________________________________________________
```



THANKS