Improving Deep Networks

TIN0255/09P9M80/D82 - Aprendizagem Profunda

2025.1 - Prof. Pedro Moura

Improving Deep Networks

- We saw in details how an individual artificial neuron works.
- Later, we arranged these neural units together as the nodes of a network.
  - Enabling the forward propagation of some input **x** through the network to produce some output **ŷ**.
- We also described how to quantify the inaccuracies of a network (compare **ŷ** to the true **y** with a cost function).
- As well as how to minimize these inaccuracies (adjust the network parameters **w** and **b** via optimization with SGD and backprop).

Improving Deep Networks

- We will now see common barriers to the creation of high-performing neural networks and techniques that overcome them.
- We will apply these ideas directly in code while architecting our **first deep neural network**.
- We'll see if we can outperform the handwritten-digit classification accuracy of our simpler, shallower architectures.

Weight Initialization

- We saw the concept of **neuron saturation**.
  - Very low or very high values of **z** diminish the capacity for a given neuron to learn.
- At the time, we offered cross-entropy cost as a solution.
- Although cross-entropy does effectively attenuate the effects of neuron saturation, **pairing it with thoughtful weight initialization will reduce the likelihood of saturation occurring in the first place**.
- Modern **weight initialization** provided a significant leap forward in deep learning capability.
  - It is one of the landmark theoretical advances between LeNet-5 and AlexNet.
  - **It broadened the range of problems artificial neural networks could reliably solve.**

Weight Initialization

- We will play around with several weight initializations to develop an intuition around how they're so impactful.
- We saw that the parameters **w** and **b** are initialized with random values.
- A network's starting approximation of **y** will be far off the mark, thereby leading to a high initial cost **C**.

Weight Initialization

- Keras by default constructs TensorFlow models that are initialized with sensible values for **w** and **b**.

- However, it's worthwhile discussing this initialization in order to understand how neural network training works.
  - And to be aware of another method for avoiding neuron saturation.

- Although Keras does a sensible job of **choosing default values** (that's a benefit of using it) → it's possible, and sometimes necessary, **to change these defaults to suit your problem**.

- Let's check out our notebook called *Weight Initialization*.

Weight Initialization

- In this notebook, we simulate 784 pixel values as inputs to a single dense layer of artificial neurons.
- The inspiration behind our simulation of these 784 inputs comes of course from our beloved MNIST digits.
- For the number of neurons in the dense layer (256), we picked a number **large enough** so that, when we make some plots later on, they **have ample data.**

Weight Initialization

- Let's now take a look at the initialization of the network parameters **w** and **b**.

- Before we begin passing training data into our network, we'd like to start with reasonably scaled parameters. Two reasons:
  - Large **w** and **b** values tend to correspond to larger **z** values and therefore saturated neurons.
  - Large parameter values would imply that the network has a **strong opinion** about how **x** is related to **y**, but **before any training on data has occurred**.

Weight Initialization

- On the other hand, parameter values of zero imply the **weakest opinion** on how **x** is related to **y**.
- We then need to look for a middle-of-the-road approach that starts training from a **balanced and learnable beginning**.
- With this in mind, when we design our neural network architecture, we select the `Zeros()` method for initializing the neurons of our dense layer with **b** = 0:
- `b_init = Zeros()`

Weight Initialization

- Following this line of thinking, we might be tempted to think that we should also initialize our network weights **w** with zeros as well.

- **This would be a training disaster:**
  - If all weights and biases were identical, many neurons in the network would treat a given input **x** identically → **we lose the benefit of having different nodes in a layer recognize different features of the input.**

  - This would give SGD a **minimum of heterogeneity** for identifying individual parameter adjustments that might reduce the cost **C**.

- More productive is to use **Symmetry Breaking**: initialize weights with a **range of different values** so that each neuron treats a given **x** uniquely, thereby providing SGD with a **wide variety of starting points** for approximating **y**.

Weight Initialization

- By chance, some of the initial neuron outputs may contribute in part to a sensible mapping from **x** to **y**.
  - This contribution will be weak at first.
  - SGD can **experiment** with it to determine whether it might contribute to a **reduction in the cost C** between the predicted **ŷ** and the target **y**.
- The vast majority of the parameters in a typical network are weights; **relatively few are biases**.
- Thus, it's acceptable (indeed, it's the most common practice) to initialize **biases with zeros**, and the **weights with a range of values near zero**.
- One straightforward way to generate **random values near zero** is to sample from the **standard normal distribution**

Weight Initialization

```
w_init = RandomNormal(stddev=1.0)
```

- To observe the impact of weight initialization, we design a neural network architecture for our single dense layer of sigmoid neurons.

```
model = Sequential()
model.add(Dense(n_dense,
                input_dim=n_input,
                kernel_initializer=w_init,
                bias_initializer=b_init))
model.add(Activation('sigmoid'))
```

- For simplicity when updating it later in this section, we add the sigmoid activation function to the layer separately by using `Activation('sigmoid')`.

Weight Initialization

- With our network set up, we use the NumPy `random()` method to generate 784 “pixel values,” which are floats randomly sampled from the range `[0.0; 1.0)`:

- `x = np.random.random((1,n_input))`

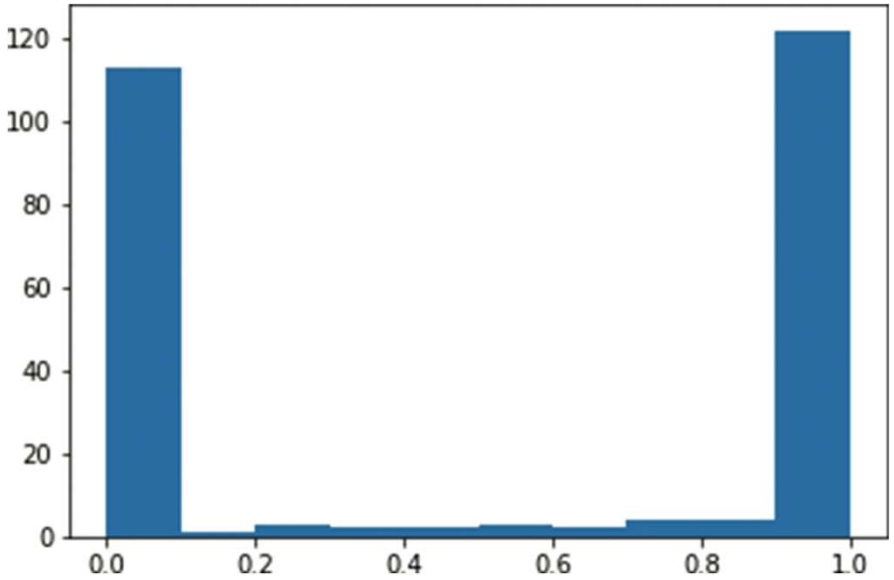
- We subsequently use the `predict()` method to forward propagate `x` through the single layer and output the activations `a`:

- `a = model.predict(x)`

- We then use a histogram to visualize the `a` activations:

- `_ = plt.hist(np.transpose(a))`

Weight Initialization



Histogram of the **a** activations output by a layer of sigmoid neurons, with weights initialized using a standard normal distribution.

Weight Initialization

- As expected, the **a** activations output from our sigmoid layer of neurons is constrained to a range from 0 to 1.

- What is undesirable about these activations → they are chiefly pressed up against the **extremes of the range**:
  - Most of them are either immediately adjacent to 0.
  - Or immediately adjacent to 1.

- This indicates that with the normal distribution from which we sampled to initialize the **layer's weights w**, we ended up encouraging our artificial neurons to **produce large z values**.

Weight Initialization

- This is unwelcome for two reasons:
  - It means the vast majority of the neurons in the layer are saturated.
  - It implies that the neurons have strong opinions about how **x** would influence **y** prior to any training on data.

- Thankfully, this ickiness can be resolved by initializing our network weights with values sampled from **alternative distributions**.

Weight Initialization

- Generally, it can be shown the **variance of the outputs linearly scales with the number of inputs.**
- The standard deviation scales with the square root of the number of inputs.
- **Solution: each weight is initiliazed to a value drawn from a Gaussian distribution with standard deviation $\sqrt{1/r}$, where r is the number of inputs that neuron.**
- Bias neurons are always initialized to zero weight.
- **Alternatively: initialize the weights to a value that is uniformly distributed in $[-1/\sqrt{r}, 1/\sqrt{r}]$.**

Xavier Glorot Distributions

- In deep learning circles, popular distributions for sampling weight-initialization values were devised by Xavier Glorot and Yoshua Bengio.

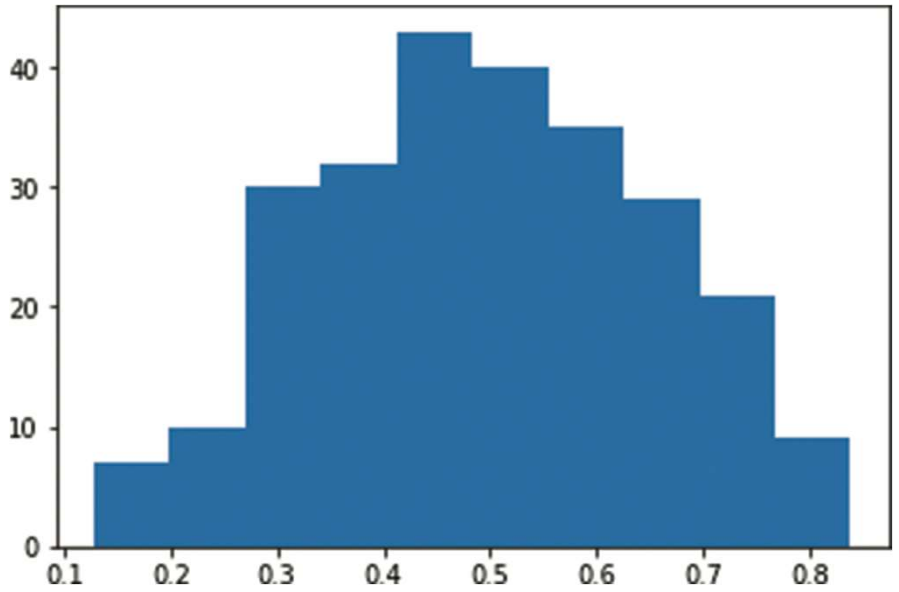
- These **Glorot distributions** (also called **Xavier distributions)** are tailored such that sampling from them will lead to **neurons initially outputting small z values**.

- The Glorot normal distribution is a truncated normal distribution.
  - It is centered at 0.
  - It has a standard deviation of $\sqrt{\frac{2}{n_{in}+n_{out}}}$, where n_{in} is the number of neurons in the preceding layer and n_{out} is the number of neurons in the subsequent layer.

Xavier Glorot Distributions

- Let's examine them in action through replacing the standard-normal-sampling code of our *Weight Initialization*.
- `w_init = glorot_normal()`
- We sample from the **Glorot normal distribution** instead.
- By restarting and rerunning the notebook, we should now observe a distribution of the activations **a** similar to the following histogram:

Xavier Glorot Distributions



Histogram of the **a** activations output by a layer of sigmoid neurons, with weights initialized using the Glorot normal distribution.

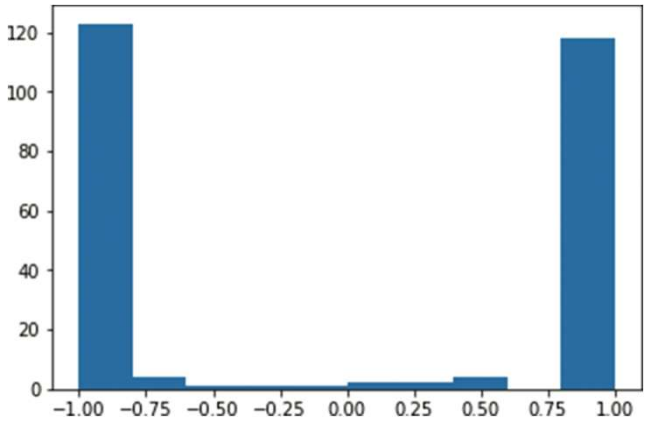
Xavier Glorot Distributions

- In stark contrast to the previous one, the **a** activations obtained from our layer of sigmoid neurons is now **normally distributed** with a **mean of ~0.5** and **few (if any) values at the extremes of the sigmoid range** (less than 0.1 or greater than 0.9).

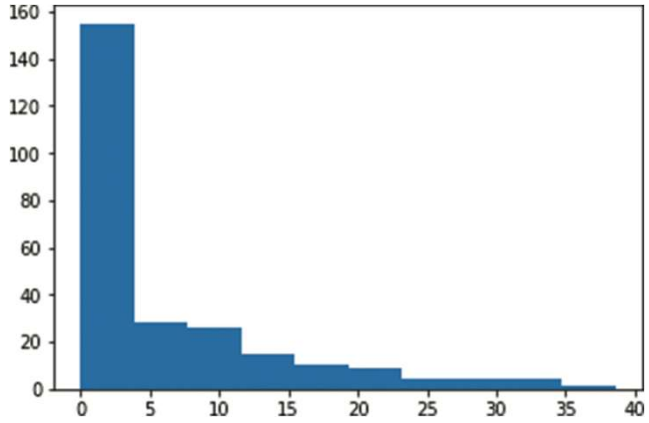
- This is a good starting point for a neural network because:
  - Few, if any, of the neurons are saturated.
  - It implies that the neurons generally have weak opinions about how **x** would influence **y**, which - prior to any training on data - is sensible.

Xavier Glorot Distributions

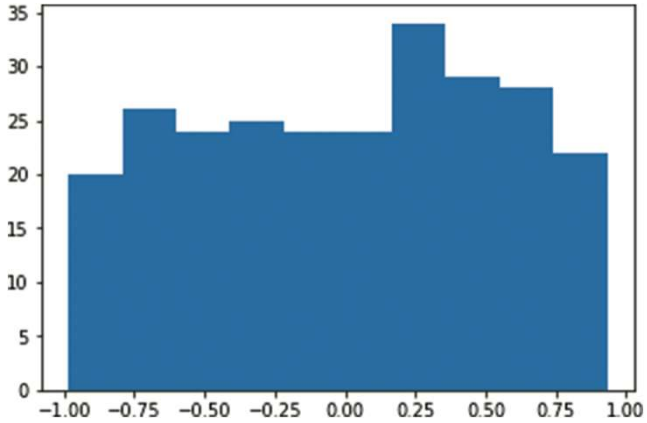
- In addition to the Glorot normal distribution, there is also the **Glorot uniform distribution**.
- The impact of selecting one of these Glorot distributions over the other when initializing your model weights is generally unnoticeable.
- Let's rerun our notebook while sampling values from the **Glorot uniform distribution** by setting
- `w_init = glorot_uniform()`



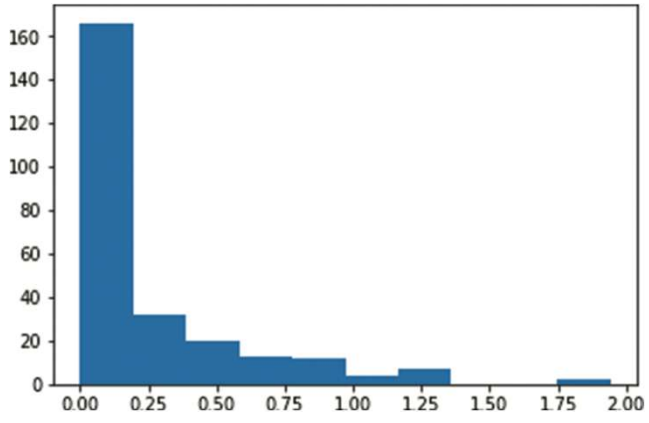
(a) tanh with standard normal init.



(b) ReLU with standard normal init.



(c) tanh with Glorot init.



(d) ReLU with Glorot init.

The activations output by a dense layer of 256 neurons, while varying activation function (tanh or ReLU) and weight initialization (standard normal or Glorot uniform).

Xavier Glorot Distributions

- Let's swap out the sigmoid activation function with tanh `(Activation('tanh'))` or ReLU `(Activation('relu'))` in the *Weight Initialization* notebook.

- We are going to observe the consequences of initializing weights with values sampled from a **standard normal distribution** relative to a Glorot distribution across a range of activations.

- Regardless of the **chosen activation function**, weight initialization with the standard normal leads to **a** activation outputs that are extreme.

- This doest not happen when initializing with Glorot.

Xavier Glorot Distributions

- One can delve into the library's documentation on a layer-by-layer basis to be sure you are aware of the parameter initialization approach used by Keras.
- Its default configuration is typically to initialize biases with 0 and to initialize weights with a Glorot distribution.
- Glorot initialization is **probably the most popular technique for initializing weights**, but there are other sensible options such as He initialization and LeCun initialization.

Lottery Ticket Hypothesis

- Jonathan Frankle; Michael Carbin. **The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks**. ICLR 2019.

- **The Lottery Ticket Hypothesis:** *A randomly-initialized, dense neural network contains a subnetwork that is initialized such that—when trained in isolation - it can match the test accuracy of the original network after training for at most the same number of iterations.*
  - The winning tickets we find have won the **initialization lottery**: their connections have initial weights that make training particularly effective.

- **Identifying winning tickets:** A winning ticket is identified by **training a network** and **pruning its smallest-magnitude weights**.

- Remaining (unpruned) connections constitute the **architecture of the winning ticket**.

Lottery Ticket Hypothesis

- Novelty: each unpruned connection's value is then **reset to its initialization** from original network **before** it was trained.

- Idea:
  - 1. Randomly initialize a neural network *f*(*x*; θ) (where $\theta_0 \sim D_\theta$).
  - 2. Train the network for *j* iterations, arriving at parameters θ_j.
  - 3. Prune *p%* of the parameters in θ_j, creating a mask *m*.
  - 4. Reset the remaining parameters to their values in θ_0, creating the winning ticket *f*(*x*; m $\cdot$ θ_0).

Lottery Ticket Hypothesis

- This pruning approach is **one-shot**: the network is trained once, *p%* of weights are pruned, and the surviving weights are reset.
- They focus on **iterative pruning**, which repeatedly:
  - Trains;
  - Prunes; and
  - Resets the network over *n* rounds.
- Each round prunes $p^{\frac{1}{n}}\%$ of the weights that survive the previous round.
- Results show that iterative pruning finds winning tickets that **match the accuracy of the original network at smaller sizes** than does one-shot pruning.

Lottery Ticket Hypothesis

- They identify winning tickets in a fully-connected architecture for MNIST and convolutional architectures for CIFAR10 across several optimization strategies (SGD, momentum, and Adam) with techniques like dropout, weight decay, batchnorm, and residual connections.

Unstable Gradients

- Another issue associated with neural networks, and one that becomes especially perturbing as we add **more hidden layers**, is **unstable gradients**.
- Unstable gradients can either be **vanishing** or **explosive** in nature.
- We will see both varieties, and then discuss a solution to these issues called **batch normalization**.

Vanishing Gradients

- When using the cost **C** between the network's predicted **ŷ** and the true **y**, backpropagation works from the output layer toward the input layer.
- It adjusts network parameters with the aim of **minimizing cost**.
- Each of the parameters is adjusted in proportion to its gradient with respect to cost:
  - If the gradient of a parameter (with respect to the cost) is **large** and **positive**, this implies that the parameter contributes a **large amount to the cost**, and so **decreasing** it proportionally would correspond to a **decrease in cost**.

Vanishing Gradients

- In the hidden layer that is **closest to the output layer**, the relationship between its parameters and cost is **the most direct**.
- The **farther away** a hidden layer is from the output layer → the **more muddled** the relationship between its parameters and cost becomes.
- As we move from the **final hidden layer toward the first hidden layer**, the gradient of a given parameter relative to cost **tends to flatten and it gradually vanishes**.
- **Consequence: the farther a layer is from the output layer, the more slowly it tends to learn.**

Vanishing Gradients

- Because of the **vanishing gradient problem**:
- Naïvely add more and more hidden layers to a neural network → eventually the hidden layers **farthest from the output would not be able to learn** to any extent.
- It cripples the capacity for the network as a whole to learn to approximate **y** given **x**.

Exploding Gradients

- They occur much less frequently than vanishing gradients
- Certain network architectures (e.g., recurrent networks) can induce **exploding gradients**.
- Gradient between a given parameter relative to cost becomes **increasingly steep** as we move from the final hidden layer toward the first hidden layer.
- Exploding gradients can also inhibit a network's capacity to learn by **saturating** the neurons with **extreme values**.

Batch Normalization

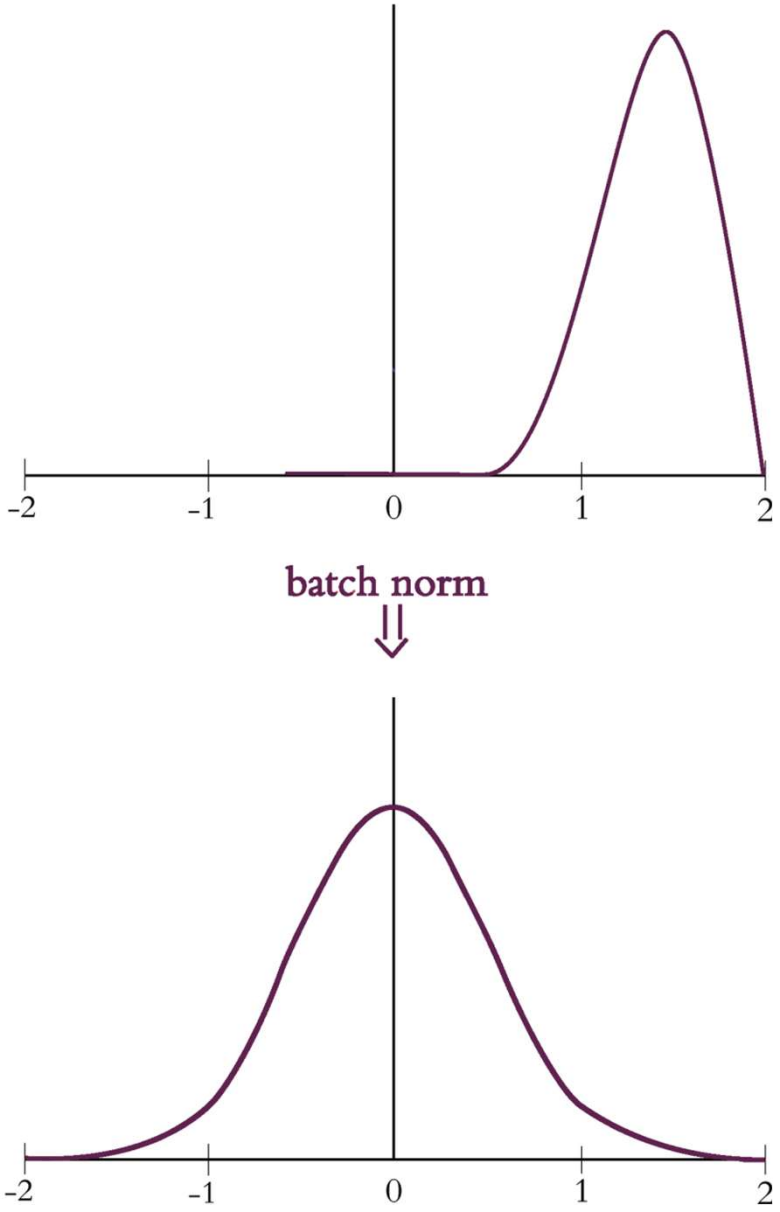
- During training, the distribution of hidden parameters in a layer may gradually move around; this is known as **internal covariate shift**.
- This is sort of the point of training → we want the parameters to change in order to learn things about the **underlying data**.
- As the distribution of the weights in a layer changes, the inputs to the next layer might be **shifted away** from an **ideal** (i.e., normal) **distribution**.
- Enter **batch normalization** (or **batch norm** for short).

Batch Normalization

- **Batch norm** takes the **a** activations output from the preceding layer, **subtracts the batch mean**, and **divides by the batch standard deviation**.
- This acts to recenter the distribution of the **a** values with a mean of 0 and a standard deviation of 1.
- Thus, if there are any extreme values in the preceding layer, **they won't cause exploding or vanishing gradients** in the next layer.

Batch Normalization

- In addition, batch norm has the following positive effects:
- It allows layers to learn more independently from each other, because **large values in one layer won't excessively influence the calculations in the next layer**.
- It allows for selection of a **higher learning rate** - because there are no extreme values in the normalized activations - thus enabling **faster learning**.
- The layer outputs are **normalized to the batch mean and standard deviation**, and that adds a **noise element** (especially with smaller batch sizes), which, in turn, contributes to **regularization**.
  - Regularization helps a network generalize to data it hasn't encountered previously, which is a good thing.



Batch normalization transforms the distribution of the activations output by a given layer of neurons toward a standard normal distribution.

Batch Normalization

- Batch normalization adds two extra learnable parameters to any given layer it is applied to: **ɣ (gamma) and β (beta)**.
- In the final step of batch norm, the outputs are **linearly transformed** by **multiplying by ɣ** and **adding β**, where ɣ is analogous to the standard deviation, and β to the mean.
  - This is the exact inverse of the operation that normalized the output values in the first place!
- However, the output values were originally normalized by the **batch mean** and **batch standard deviation**, whereas ɣ and β are learned by SGD.
- We initialize the batch norm layer with ɣ = 1 and β = 0 → at the start of training this linear transformation **makes no changes**; batch norm is allowed to normalize the outputs as intended.

Batch Normalization

- As the network learns → it may determine that **denormalizing** any given layer's activations is optimal for reducing cost.
- If batch norm is not helpful the network **will learn to stop** using it on a layer-by-layer basis.
- Because ɣ and β are continuous variables, the network can decide to **what degree** it would like to denormalize the outputs, depending on what **works best to minimize the cost**.

Model Generalization (Avoiding Overfitting)

- Previously, we saw that after training a model for a certain number of epochs:
  - Cost calculated on the **validation dataset** - which may have been decreasing nicely over earlier epochs - could begin to **increase**.
  - Despite the fact that the cost calculated on the **training dataset** is still **decreasing** → this the cost we are **minimizing**.

- This situation - when **training cost** continues to go **down** while **validation cost** goes **up** - is formally known as **overfitting**.

Model Generalization (Avoiding Overfitting)

- Let's illustrate this concept through the visualization of a Figure containing data points scattered along the **x** and **y** axes.
  - There is some underlying distribution that describes these points and we show a sampling from that distribution.

- Our goal is to generate a model that explains the relationship between **x** and **y**.

- Most importantly, one that also **approximates** the original distribution → model will be able to **generalize** to new data points drawn from the distribution.
  - Not just model the sampling of points we already have.

Model Generalization (Avoiding Overfitting)

- Let's illustrate this concept through the visualization of a Figure containing data points scattered along the **x** and **y** axes.
  - There is some underlying distribution that describes these points and we show a sampling from that distribution.

- Our goal is to generate a model that explains the relationship between **x** and **y**.

- Most importantly, one that also **approximates** the original distribution → model will be able to **generalize** to new data points drawn from the distribution.
  - Not just model the sampling of points we already have.

Model Generalization (Avoiding Overfitting)

- Straight line → **underfitting**.
  - The cost is high and the model would not generalize well to new data points.
  - Model is not complex enough.

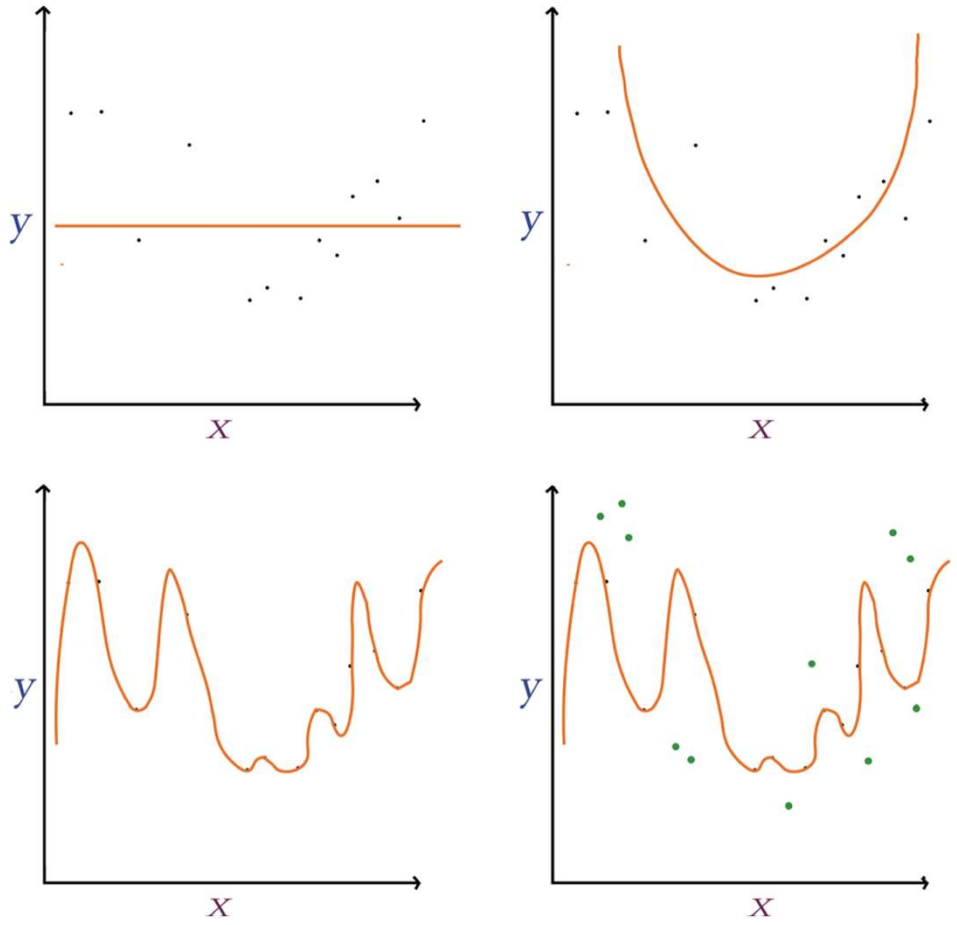
- Model with two parameters → parabola-shaped curve to the data
  - The cost is much lower relative to the linear model, and it appears the model would also generalize well to new data.

- Model with too many parameters (more parameters than we have data points).
  - We reduce the cost associated with our training data points to nil (**there is no gap between the curve and data**).
  - The model fits these validation data poorly and so it results in a significant validation cost.
  - **This is overfitting**.

Model Generalization (Avoiding Overfitting)

- It is a perfect model for the training data, but it doesn't actually capture the relationship between **x** and **y** well.

- It has learned the exact features of the training data too closely, and it subsequently performs poorly on unseen data.



Fitting *y* given *x* using models with varying numbers of parameters. Top left: A single-parameter model underfits the data. Top right: A two-parameter model fits a parabola that suits the relationship between *x* and *y* well. Bottom left: A many-parameter model overfits the data, generalizing poorly to new data points.

Model Generalization (Avoiding Overfitting)

- It should not be surprising that deep learning architectures regularly have millions of parameters.
- Working with datasets that have such a large number of parameters but with perhaps only thousands of training samples could be a recipe for severe overfitting.
- We want to capitalize on deep, sophisticated network architectures even if we don't have oodles of data at hand.
- **Thankfully we can rely on techniques specifically designed to reduce overfitting.**

L1 and L2 Regularization

- In branches of machine learning other than deep learning: use of **L1 and L2 regularization** to reduce overfitting is prevalent.
- Alternatively known as **LASSO (Least Absolutely Shrinkage and Selection Operator) Regression** and **Ridge Regression**.
- Both penalize models for including parameters by adding the parameters to the model's cost function.
- The larger a given parameter's size, the more that parameter adds to the cost function.

L1 and L2 Regularization

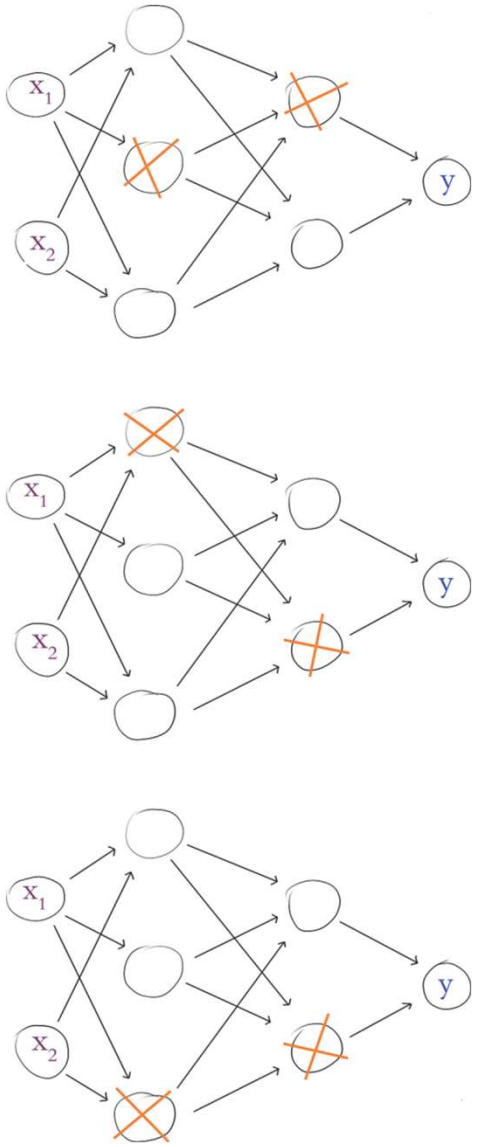
- Because of this, parameters are not retained by the model unless they appreciably contribute to the reduction of the difference between the model's estimated $\hat{y}$ and the true y.
- In other words, extraneous parameters are pared away.
- **Difference between L1 and L2:** L1's additions to cost correspond to the absolute value of parameter sizes, whereas L2's additions correspond to the square of these.

Dropout

- L1 and L2 regularization work fine to reduce overfitting in deep learning models.
- However, deep learning practitioners tend to favor a technique tailored for deep learning: **dropout**.
  - Intuitive yet powerful concept for preventing **overfitting**.
- This technique was developed by Geoff Hinton and his colleagues at the University of Toronto.
- It was made famous by its incorporation in their benchmark-smashing AlexNet architecture.

Dropout

- Dropout simply **pretends** that a randomly selected proportion of the neurons in each layer **don't** exist during each round of training.
- To illustrate this, let's see an example showing three rounds of training.
- For each round, we remove a specified proportion of hidden layers by random selection:
  - For the first hidden layer → we configured it to **drop out 33%** of the neurons.
  - For the second hidden layer → we configured it to **drop out 50%** of the neurons.



Three rounds of training with dropout.

Dropout

- There is no “memory” of which neurons have been dropped out on previous training rounds.
- It is **by chance** alone that the neurons dropped out in the second round are **distinct** from those dropped out in the first.
- **The second neuron of the second hidden layer was randomly selected in the last two rounds of training.**

Dropout

- Dropout doesn't (directly) constrain how large a given parameter value can become.
- It is an effective regularization technique, because it **prevents any single neuron** from becoming excessively influential within the network.
- Dropout makes it challenging for some very specific aspect of the training dataset to create an overly specific forward-propagation pathway through the network.
  - **On any given round of training, neurons along that pathway could be remove.**
- **The model doesn't become overreliant on certain features of the data to generate a good prediction.**

Dropout

- When validating a model that was trained using dropout (or when making real-world inferences with such model) → we must take an extra step first.
- During validation or inference, we would like to **leverage the power** of the full network: **its total complement of neurons**.
- Problem: we only ever used a subset of the neurons to forward propagate **x** through the network and estimate $\hat{\mathbf{y}}$.
- Naïvely carry out this forward propagation with suddenly **all** of the neurons → our $\hat{\mathbf{y}}$ would emerge disoriented.

Dropout

- There are now too many parameters → totals after calculations would be larger than expected.
- Compensate for the additional neurons → we must correspondingly adjust our neuron parameters downward.
- **Drop out half of the neurons** in a hidden layer durining training → we need to **multiply the layer's parameters by 0.5** prior to validation or inference.
- For a hidden in which we dropped out 33.3% of the neurons during training → we must multiply the layer's parameters by 0.667 prior to validation.
- **If the probability of a given neuron being retained during training is *p*, then we multiply that neuron's parameters by (1 - *p*) prior to carrying out model validation or inference.**

Dropout

- Thankfully, Keras handles this parameter-adjustments process for us automatically.

- However, when working in other deep learning libraries (e.g., low-level TensorFlow), you may need to be mindful and remember to carry out these adjustments yourself.

- **Ensembles of Statistical Models:** dropout produces an ensemble.
  - During each round of training, a random subnetwork is created and its parameter values are tuned.
  - At the conclusion of training, all of these subnetworks are reflected in the parameter values throughout the final network.
  - The final network is **an aggregated ensemble of its constituent subnetworks**.

Dropout

- Network architecture options pertaining to dropout are hyperparameters.

- Rules of thumb (which layers and how much to apply):
  - If your network is **overfitting** to your training data → then dropout is warranted somewhere in the network.
  - Even if your network isn't obviously overfitting to your training data, adding some dropout to the network may improve validation accuracy - especially in later epochs of training.
  - Applying dropout to **all** of the hidden layers in your network **may be overkill**. If your network has a fair bit of depth, it may be sufficient to apply dropout solely to later layers in the network (the earliest layers may be harmlessly identifying features).
    - Begin by applying dropout only to the final hidden layer and observing whether this is sufficient for curtailing overfitting; if not, add dropout to the next deepest layer, test it, and so on.

Dropout

- If your network is struggling to reduce validation cost or to recapitulate low validation costs attained when less dropout was applied, then you've added too much dropout - pare it back.

- How much dropout?
  - **Machine vision:** Dropping out 20 percent up to 50 percent of the hidden-layer neurons tends to provide the highest validation accuracies.

  - **In natural language applications:** where individual words and phrases can convey particular significance, we have found that dropping out a smaller proportion - between 20 percent and 30 percent of the neurons in a given hidden layer - tends to be optimal.

Data Augmentation

- In addition to regularizing your model's parameters to reduce overfitting, another approach is to increase the size of your training dataset.

- Possibility of collecting additional high-quality training data for the problem you're working on → then you should do so!
  - **It could be a problem to label this collected data.**

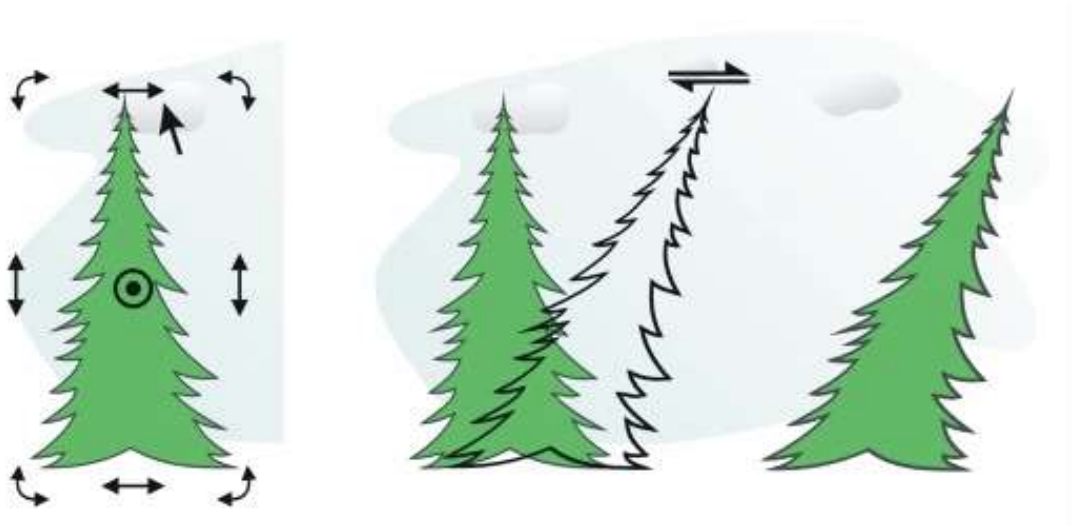
- The **more data provided** to a model during training, **the better the model** will be able to **generalize to unseen** validation data.

Data Augmentation

- Generally, collecting fresh data is a pipe dream.
- It may be possible to generate new training data from existing data by **augmenting** it, **artificially expanding** your training dataset.
- With the MNIST digits, for example, many different types of transforms can be applied to create suitable handwritten digits:
  - Skewing the image
  - Blurring the image
  - Shifting the image a few pixels
  - Applying random noise to the image
  - Rotating the image slightly

Data Augmentation

- Skewing the image



Data Augmentation

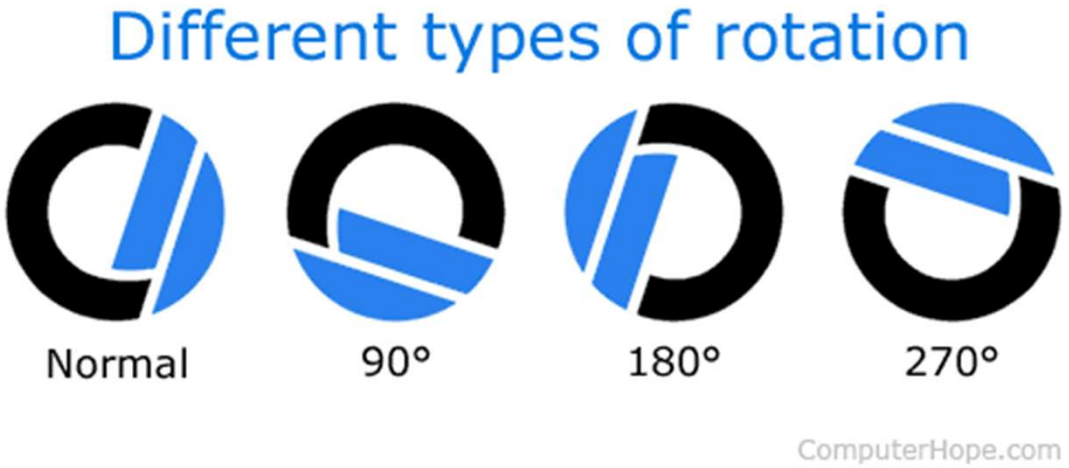
- Blurring the image

Data Augmentation

- Applying random noise to the image:

Data Augmentation

- Rotating the image:



Data Augmentation

- As can be checked on the website of Yann LeCun, many of the record-setting MNIST validation dataset classifiers use data augmentation.

- It is specially useful for problems for which we dispose of few data (e.g., images of medical exams).

Data Augmentation

- Data Augmentation can also be applied to text, although this kind of approach is more recent.
- For simplicity, let's assume our text corresponds to a single sentence.
- Possible operations:
  - Random exchange of two words.
  - Replace a random word with its synonym.
  - Random deletion of a word.
  - Random insertion of a word.
  - Random deletion of a character in a word.
  - Back translation to the original language.
  - And many more.

Data Augmentation

- **Problem → there is no API to perform these tasks for the portuguese language.**
- **Possibility → the use of ChatGPT to perform these operations.**

Fancy Optimizers

- So far, we have only used one optimization algorithm: stochastic gradient descent (SGD).
- Although it performs well, researchers have devised ways to improve it.

Momentum

- The first SGD improvement was to consider **momentum**. Analogy:
  - It's winter and our trilobite is skiing down a snowy gradient mountain.
  - A local minimum is encountered.
  - Momentum of the trilobite's movement down the slippery hill will keep it moving.
  - Minimum will be easily bypassed.
  - **The gradients on previous steps have influenced the current step.**

- Calculate momentum in SGD → take a moving average of the gradients for each parameter and use that to update the weights in each step.

- With this, we have the additional **parameter β (beta)**, which ranges from 0 to 1, and which controls **how many previous gradients are incorporated in the moving average**.

Momentum

- **Small β values permit older gradients to contribute to the moving average (this can be unhelpful).**
  - Trilobite wouldn't want the steepest part of the hill to guide its speed as it approaches the lodge for its après-ski drinks.

- **Typically we'd use larger β values, with β = 0.9 serving as a reasonable default.**

Nesterov Momentum

- Another version is **Nesterov Momentum**.
- **Moving average** of the gradients is first used to **update the weights** and find the gradients at **whatever that position** may be.
  - This is equivalent to a quick peek at the position where momentum might take us.
- We then use the gradients from this sneak-peek position to execute a gradient step **from our original position**.
- In other words: trilobite is suddenly aware of its speed down the hill, so it's taking that into account, guessing where its own momentum might be taking it, and **then adjusting its course before it even gets there**.

AdaGrad

- Both momentum approaches improved SGD.
- However, they both use a single learning rate η for all parameters.
- We could think of an individual learning rate for each parameter.
  - Enabling those parameters that have already reached their optimum to slow or halt learning.
  - While those that are far from their optima can keep going.
- **That's exactly what can be achieved with the other optimizers we'll see now: AdaGrad, AdaDelta, RMSProp, and Adam.**

AdaGrad

- The name AdaGrad comes from “adaptive gradient.”
- In this variation, **every parameter has a unique learning rate** that scales depending on the importance of that feature.
- This is especially useful for sparse data where some features occur only rarely: **when those features do occur, we’d like to make larger updates of their parameters.**
- **It maintains a matrix of the sum of squares of the past gradients for each parameter, and divides the learning rate by its square root.**

AdaGrad

- AdaGrad is the first introduction to the parameter ϵ (epsilon), which is a doozy: it is a **smoothing factor** to **avoid divide-by-zero errors** and can safely be left at its default value of $\epsilon = 1 \times 10^{-8}$.

- A significant benefit of AdaGrad is that it minimizes the need to tinker with the learning rate hyperparameter η.
  - You can generally just set-it-and-forget-it at its default of $\eta = 0.01$.

- Considerable downside of AdaGrad:
  - Matrix of past gradients can increase in size.
  - Learning rate is then increasingly divided by a larger and larger value.
  - This eventually renders the learning rate impractically small and so learning essentially stops.

AdaDelta and RMSProp

- AdaDelta resolves the gradient-matrix-size shortcoming of AdaGrad by maintaining a **moving average** of previous gradients.
  - In the same manner that momentum does.
- AdaDelta also eliminates the η term, so a learning rate doesn't need to be configured at all.

AdaDelta and RMSProp

- RMSProp (root mean square propagation) was developed by Geoff Hinton at about the same time as AdaDelta.
- It works similarly except it retains the learning rate η parameter.
- Both RMSProp and AdaDelta involve an extra hyperparameter ρ, or decay rate.
  - It is analogous to the β value from momentum.
  - It guides the size of the window for the moving average.
- Recommended values for the hyperparameters are:
  - ρ = 0.95 for both optimizers.
  - η = 0.001 for RMSProp.

Adam

- Nowadays, Adam is the most used optimizer in the book and in applications.
- Adam stands for Adaptive Moment Estimation.
  - It builds on the optimizers that came before it.
- It is essentialy RMSProp with two exceptions:
- **An extra moving average is calculated: past gradients for each parameter**
  - It is used to inform the update instead of the the actual gradients at that point.
- **Bias trick: used to help prevent these moving averages from skewing toward zero at the start of training.**

Adam

- Adam has two β hyperparameters:
  - One for each of the moving averages that are calculated.
  - Recommended defaults are $\beta_1 = 0.9$ and $\beta_2 = 0.999$.
- Learning rate default with Adam is $\eta = 0.001$.
  - We can generally leave it there.

Adam

- RMSProp, AdaDelta, and Adam are similar → they can be used interchageably in similar applications.
  - Bias correction may help Adam later in training.
- These new optimizers are in vogue, but there is still a strong case for simple **SGD with momentum** (or **Nesterov momentum**).
  - In some cases it performs better.
- As we already know, many things in Deep Learning are highly empirical.
  - We **should experiment with optimizers** and observe what **works best** for a particular model architecture and problem.

Regression

- When we discussed Supervised Learning, we mentioned that these can involve either **classification** or **regression**.
- Nearly all our models are used for classifying inputs into one category or another.
- Now we are going to see how to adapt neural network models to **regression tasks**.
  - Any problem where you'd like to predict some continuous variables.

Regression

- Examples of regression problems:
  - Predicting the future price of a stock.
  - Forecasting how many centimeters of rain may fall tomorrow.
  - Modeling how many sales to expect of a particular product.
- We are going apply deep learning for a classic dataset to estimate the price of housing near Boston, Massachusetts, in the 1970s.
- Let's take a look at the *Regression in Keras* notebook.

Regression

- The only unfamiliar dependency is the `boston_housing` dataset.
  - Conveniently bundled into the Keras library.
- Loading the data is as simple as with the MNIST digits:

```
(X_train, y_train), (X_valid, y_valid) = boston_housing.load_data()
```

- Calling the `shape` parameter of `X_train` and `X_valid`, we find that there are 404 training cases and 102 validation cases.

Regression

- For each case - a distinct area of the Boston suburbs – we have 13 predictor variables related to:
  - building age, mean number of rooms, crime rate, the local student-to-teacher ratio, and so on.
- The median house price (in thousands of dollars) for each area is provided in the **y** variables.
- As an example, the first case in the training set has a median house price of $15,200.32.

Regression

- With only 13 input values and a few hundred training cases we would gain little from a deep neural network with oodles of neurons in each layer.
- **It would be better to opt for a two-hidden-layer architecture consisting of merely 32 and 16 neurons per layer.**
- We applied batch normalization and a touch of dropout to avoid overfitting to the particular cases of the training dataset.
- **Most critically**, in the output layer we set the `activation` argument to `linear`.
  - The option to go with when you'd like to predict a **continuous variable**, as we do when **performing regression**.

Regression

- Linear activation function outputs **z** directly so that the network's **ŷ** can be any numeric value (e.g., dollars, centimeters).
  - Instead of being squashed into a probability between 0 and 1 (as happens with sigmoid or softmax).
- When compiling the model, another regression-specific adjustment we made is using mean squared error (MSE) in place of cross-entropy:
- `model.compile(loss='mean_squared_error', optimizer='adam')`
- Cross-entropy cost function is specifically designed for classification problems, in which **ŷ** is a probability.
- For regression problems, where the output is inherently **not a probabilty**, we use MSE instead.
  - Other functions applicable to regression problems: mean absolute error (MAE) and Huber loss.

Regression

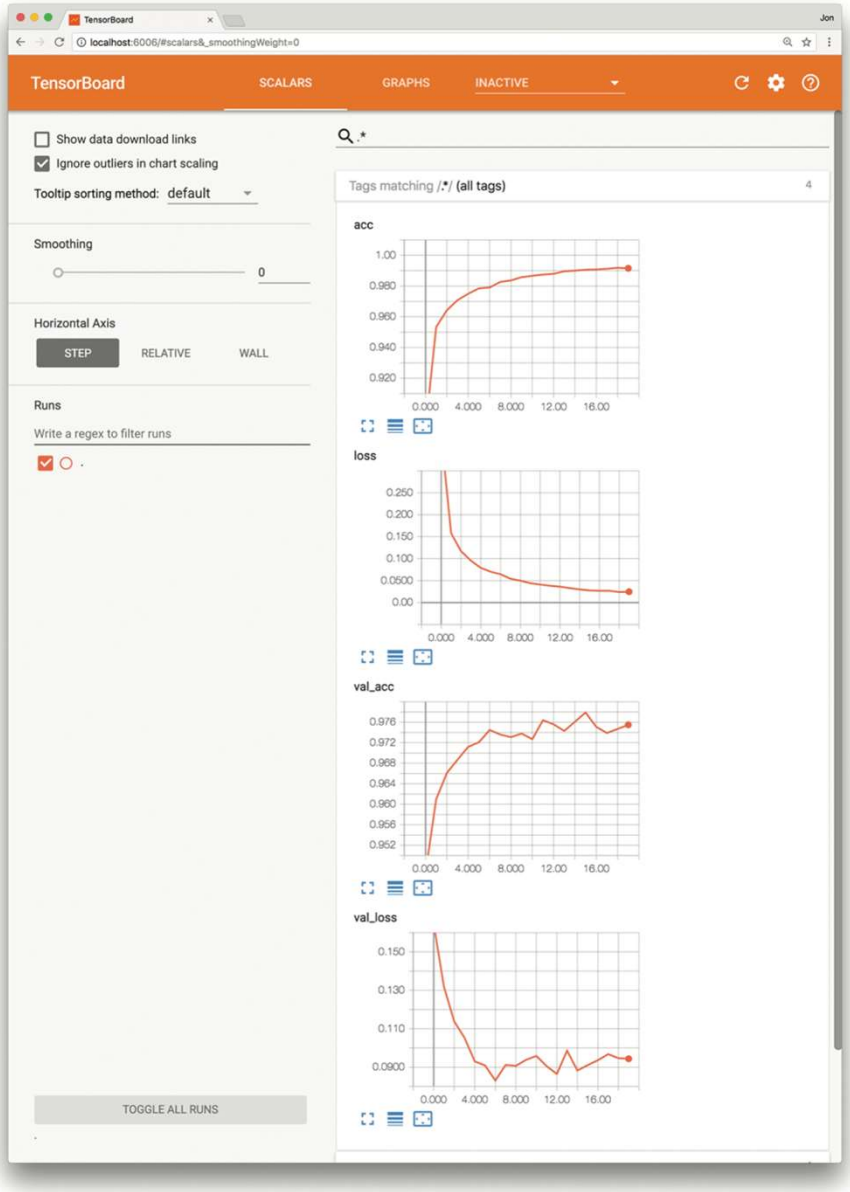
- One shall note that we left out accuracy metric when compiling.
- **There's no point in calculating accuracy**, because it isn't relevant to **continuous variables** as it is with categorical ones.
- We trained for 32 epochs because training for longer produced no lower validation losses in experiments.
- By our final (32nd) epoch, this loss had risen considerably.
- We will see how to save our model parameters after each epoch of training so that the best-performing epoch can be reloaded later.
  - For the time being, we're stuck with the relatively crummy parameters from the final epoch

TensorBoard

- When evaluating the performance of your model epoch over epoch, it can be tedious and time-consuming to read individual results numerically, as we did after running our codes, particularly if the model has been training for many epochs.

- Instead, TensorBoard is a convenient, graphical tool for this purpose:
  - Visually tracking model performance in real time.
  - Reviewing historical model performances.
  - Comparing the performance of various model architectures and hyperparameter settings applied to fitting the same data.

- TensorBoard comes automatically with the TensorFlow library.
  - Instructions for getting it up and running are available via the TensorFlow site.



The TensorBoard dashboard enables you to, epoch over epoch, visually track your model's cost (`loss`) and accuracy (`acc`) across both your training data and your validation (`val`) data.

Referências

1. Capítulo 9 do livro *“Deep Learning Illustrated: A Visual, Interactive Guide to Artificial Intelligence”* de Jon Krohn, Grant Beyleveld e Aglaé Bassens.
2. Jonathan Frankle; Michael Carbin. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. International Conference on Learning Representations 2023 (ICLR 2023).