A Proof of Theorem 3.1

In this section, we show that in the infinite width limit, warm starting a neural network will result in the same predictions as the cold started variant when trained using gradient descent for $t = \infty$. We give the proof for a feed-forward neural network, but as shown by Yang and Littwin [21], it is trivial to show that the same proof structure can be used for any other architecture.

Background We mostly use the same notation as Jacot et al. [7]. We focus on fully connected feed-forward neural networks with $L + 1$ hidden layers numbered from 0 (input) to L (output), where each layer has $n_0, \dots, n_L$ neurons. We assume that its nonlinearity is a Lipschitz, twice differentiable function $\sigma : \mathbb{R} \to \mathbb{R}$. Such a network has $P = \sum_{l=0}^{L-1} (n_l + 1)n_{l+1}$ parameters: A weight matrix $W^{(l)} \in \mathbb{R}^{n_l \times n_{l+1}}$ and a bias vector $b^{(l)} \in \mathbb{R}^{n_{l+1}}$ per layer. This network is defined as f_θ where $\theta = \cup_{l=0}^{L} \theta^l$ and $\theta^l = vec(W^{(l)}, b^{(l)})$. The function f_θ is characterized according to the definition:

$$\begin{aligned}
\alpha^{(0)}(x;\theta) &= x \\
\tilde{\alpha}^{(l+1)}(x;\theta) &= \frac{1}{\sqrt{n_l}} W^{(l} \alpha^{(l)}(x;\theta) + \beta b^{(l)} \\
\alpha^{(l)}(x;\theta) &= \sigma(\tilde{\alpha}^{(l+1)}(x;\theta)) \\
f_\theta(x) &\coloneqq \tilde{\alpha}^{(L)}(x;\theta)
\end{aligned} \tag{14}$$

We define the *realization function* of the network as $f_{(L)} : \mathbb{R}^P \to \mathcal{F}$, the function mapping parameters $\theta : \mathbb{R}^P$ to functions $f_\theta \in \mathcal{F}$ where $\mathcal{F}$ is the space of all neural networks of this architecture.

Assuming that the training set is defined as $\mathcal{D} \subseteq \mathbb{R}^{n_0} \times \mathbb{R}^{n_L}$, we define p^{in} to be the fixed empirical distribution of the finite input dataset $\{(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N)\} : \frac{1}{N} \sum_{i=0}^{N} \delta_{x_i}$. Thus, $\mathcal{F}$ is defined as all the functions $\{f : \mathbb{R}^{n_0} \to \mathbb{R}^{n_L}\}$. On this space, we consider the seminorm $|| \cdot ||_{p^{in}}$, defined as

$$\langle f, g \rangle_{p^{in}} = \mathbb{E}_{x \sim p^{in}}[f(x)^T g(x)]. \tag{15}$$

The dual space of $\mathcal{F}$ with respect to p^{in} can be defined as $\mathcal{F}^*$. $\mathcal{F}^*$ is the set of all linear forms $\mu : \mathcal{F} \to \mathbb{R}$. We can define each element of this set as $\mu = \langle d, \cdot \rangle_{p^{in}}$ for some $d \in \mathcal{F}$.

In the process of optimizing the parameters of a neural network f, we define the cost functional as: $C(f) = \frac{1}{N} \sum_{i=1}^{N} (f(x_i) - y_i)^2$. As we assumed that p^{in} is fixed, the value of $C(f)$ only depends on the values of $f \in \mathcal{F}$ at the datapoints $(x, y) \in p^{in}$. Thus, the functional derivative of the cost functional can be viewed as a member of the dual space of $\mathcal{F}$, namely $\mathcal{F}^*$. We note by $d|_{f_0} \in \mathcal{F}$ the corresponding dual element of partial derivative of the cost functional with respect to the function f at f_0, namely: $\partial_f^{in} C|_{f_0}$. Thus, $\partial_f^{in} C|_{f_0} = \langle d|_{f_0}, \cdot \rangle_{p^{in}}$.

According to Jacot et al. [7], *Kernel Gradient* $\nabla_{\mathcal{K}} C|_{f_0} \in \mathcal{F}$ is defined as $\Phi_{\mathcal{K}} \left(\partial_f^{in} C|_{f_0} \right)$ where Φ is a map from $\mathcal{F}^*$ to $\mathcal{F}$ defined as: $[\Phi_{\mathcal{K}}(\mu)]_{i,\cdot}(x) = \langle d, \mathcal{K}_{i,\cdot}(x, \cdot) \rangle_{p^{in}}$ where $\mu = \langle d, \cdot \rangle_{p^{in}}$. Here, $\mathcal{K}$ refers to a multi-dimensional kernel which is defined as a function $\mathbb{R}^{n_0} \times \mathbb{R}^{n_0} \to \mathbb{R}^{n_L \times n_L}$ such that $\mathcal{K}(x, x') = \mathcal{K}(x', x)^\top$. They showed that when trained using gradient descent on $C \circ F_{(L)}$, the neural network function's evolution in time can be captured using *kernel gradient descent* with the corresponding Neural Tangent Kernel (NTK): $\partial_t f_{\theta(t)} = -\nabla_{\mathcal{K}} C|_{f_{\theta(t)}}$ where $\mathcal{K}$ is the corresponding Neural Tangent Kernel of $f_{\theta(t)}$. For simplicity, from here onwards, we drop the θ index from a function $f_{\theta_{(t)}}$ and show it by f_t. Since one can define a map between the time t and function $f_{\theta_{(t)}}$ when the training setting is fixed, this doesn't create any confusion. Thus, the cost functional evolves as

$$\partial_t C|_{f_t} = -\langle d|_{f_t}, \nabla_{\mathcal{K}} C|_{f_t} \rangle_{p^{in}} \tag{16}$$

Here, $-d|_{f_t} \in \mathcal{F}$ defines the training direction of function $f_{(t)}$ in the function space $\mathcal{F}$ while being trained using gradient descent (flow). As mentioned earlier, when the we train using $C \circ F_{(L)}$ loss, this training direction is the dual of $\partial_f^{in} C|_{f_t}$.

Setting A. We have C overlapping datasets, $S_1, S_2, \dots, S_C$ such that $\forall i > j; S_j \subset S_i$ and $\forall i \in [1, C]; S_i = \{(x_1, y_1), \dots, (x_{m_i}, y_{M_i})\}$ where M_i is the size of i'th dataset. We initialize the network

f with parameters θ_0 such that θ_0 satisfies the initialization criteria mentioned in Jacot et al. [7] (namely, LeCun initialization). We define f_0 as f with parameters θ_0. f has $L+1$ layers such that in the limit $n_0, n_1, \ldots, n_L \to \infty$ sequentially. We train f sequentially on all the sets using gradient descent with $C \circ F_L$ loss for infinite time:

$$f_0 \xrightarrow[t=\infty]{S_1} f_{S_1} \xrightarrow[t=\infty]{S_2} f_{S_{1,2}} \to \ldots \xrightarrow[t=\infty]{S_C} f_{S_{1,2,\ldots,C}}\,. \tag{17}$$

In the following, we first note that starting from the initialization and training the network according to (17), the sum of integrals of the training directions remains stochastically bounded (Remark A.1). Thus, based on Theorem 2 in [7], the corresponding Neural Tangent Kernel of f remains asymptotically constant (and according to their Theorem 1, it converges in probability to the limiting Neural Tangent Kernel at initialization). Next, in Theorem A.2, we prove that under this setting, the resulting function of sequential warm-start training on S_1 to S_C, is the same as the function that we get when doing simple gradient descent on only the last dataset S_C when starting from initialization (Also known as cold-start). In other words, $\forall x \in \mathbb{R}^{n_0}; f_{S_{1,\ldots,C}}(x) = f_{S_C}(x)$ where f_{S_C} is derived using $f_0 \xrightarrow[t=\infty]{S_C} f_{S_C}$.

Remark A.1. *Under Setting A, assuming that we have a function $\mathcal{T}: \mathbb{R} \to \mathbb{R}$ which gets past time since starting gradient descent as the input and outputs the index i of the dataset S_i currently being trained in the sequential training process, $\lim_{T\to\infty} \int_{t=0}^{T} ||d|_{f_t}^{S_{\mathcal{T}(t)}}||_{p^{in}} dt$ is stochastically bounded.*

Proof. We start by deriving the Gâteaux derivative of the cost functional $C(f) = \frac{1}{N}\sum_{i=1}^{N} ||f(x) - y||_2^2$:

$$\partial_f^{in} C|_{f_t}(f) = \frac{2}{N}\sum_{i=1}^{N} f(x_i)^\top \left(f_t(x_i) - y_i\right) \tag{18}$$

This shows the amount of change in $C(f_t)$ when f_t is moved towards f by an infinitesimal t. As mentioned before, this functional derivative only depends on the values of f on the datapoint in p^{in}. Thus, it's in the dual space of $\mathcal{F}$, noted by $\mathcal{F}^*$ and we can write it as $\langle d|_{f_t}, f\rangle_{p^{in}}$. We're interested in deriving the closed form of $d|f_t$ in this inner product.

$$\langle d|_{f_t}, f\rangle_{p^{in}} = \frac{1}{N}\sum_{i=1}^{N} d|_{f_t}(x_i)^\top f(x_i) = \frac{2}{N}\sum_{i=1}^{N} f(x_i)^\top \left(f_t(x_i) - y_i\right) \tag{19}$$

This implies

$$\forall (x_i, y_i) \in p^{in};\; d|_{f_t}(x_i) = \frac{1}{2}\left(f_t(x_i) - y_i\right). \tag{20}$$

If we define $f^*(x)$ as the function that maps each datapoint to its label on p^{in} and is arbitrary elsewhere, $d|_{f_t}(x) = \frac{1}{2}\left(f_t(x) - f^*(x)\right)$ on p^{in}. Accordingly, when we train using only a portion of the data $S_i \subseteq p^{in}$ such that $\mathcal{X}_{S_i} = \{x : (x,y) \in S_i\}$, but the cost functional still operates on whole of p^{in}, we have that:

$$d|_{f_t}^{S_i}(x) = \frac{I(x \in \mathcal{X}_{S_i})}{2}\left(f_t(x) - f_{S_i}^*(x)\right) \tag{21}$$

where $f_{S_i}^*$ is defined on S_i as f^* is on p^{in}. To analyze $\int_{t=0}^{T} ||d|_{f_t}^{S_i}||_{p^{in}} dt$, we first derive $||d|_{f_t}^{S_i}||_{p^{in}}$:

$$||d|_{f_t}^{S_i}||_{p^{in}} = \mathbb{E}_{x\sim p^{in}}\left[\left(d|_{f_t}^{S_i}(x)\right)^2\right] = \frac{1}{4N}\sum_{x_j \in \mathcal{X}_{S_i}} ||f_t(x_j) - f_{S_i}^*(x_j)||_2^2 \tag{22}$$

As Jacot et al. [7] (Section 5) mentioned, as t grows, this norm is strictly decreasing and thus the integral is bounded. Moreover, as we're following the direction of gradient flow in the functional space, $\exists t \geq 0\;\; f_t(x_j) = f_t^*(x_j)$ for all datapoints x_j in S_i. Thus, $\lim_{T\to\infty} \int_{t=0}^{T} ||d|_{f_t}^{S_i}||_{p^{in}} dt$ is also stochastically bounded. We can apply the same structure for the case where we start from f_{S_i} and perform gradient flow towards $f_{S_{i+1}}$, showing that the integral in the infinite time limit is also stochastically bounded. Thus, it's straightforward to show that using induction, when training the neural network sequentially on $S_1, S_2, \ldots, S_C$ for sequentially infinite time, the integrals of training directions remain stochastically bounded. □

Theorem A.2. *Under Setting A, the following equality holds:*

$$\forall x \in \mathbb{R}^{n_0} \quad f_{S_{1,\dots,C}}(x) = f_{S_C}(x) \tag{23}$$

Proof. As Remark A.1 showed, when training a neural network on multiple overlapping datasets under Setting A, the training direction remains stochastically bounded. Thus, as the width of the layers of the network tend to infinite sequentially, we can use Jacot et al. [7]'s Theorem 2 to show that the NTK $\mathcal{K}$ also remains constant during training in this setting. Accordingly, when trained only on a dataset S_i, the neural network function f evolves as

$$\begin{aligned}\partial_t f_t(x) &= -\nabla_{\mathcal{K}} C|_{f_t}^{S_i}(x) = -\Phi_{\mathcal{K}}\left(\partial_f^{S_i} C|_{f_t}\right) = -\frac{1}{2N}\sum_{x_j \in S_i} \mathcal{K}(x, x_j)(f_t(x_j) - f^*_{S_i}(x_j)) \\ &= \underbrace{\frac{1}{2N}\sum_{x_j \in S_i} \mathcal{K}(x, x_j) f_t(x_j)}_{\text{function of } f_t(x),\, x \text{ at } t} - \underbrace{\frac{1}{2N}\sum_{x_j \in S_i} \mathcal{K}(x, x_j) f^*(x_j)}_{\text{function of } x \text{ at } t}\end{aligned} \tag{24}$$

where $\mathcal{K}$ is the corresponding NTK of f_t, which remains constant during training. If we look closely, we witenss that this is a system of ODEs, whose solution for the finite dataset S_i from initialization f_0 can be written as

$$f_t(\mathcal{X}_{S_i}) = f^*_{S_i}(\mathcal{X}) + e^{-t\mathcal{K}_{S_i}}\left(f_0(\mathcal{X}_{S_i}) - f^*_{S_i}(\mathcal{X}_{S_i})\right). \tag{25}$$

where $\mathcal{K}_{S_i} = \mathcal{K}(\mathcal{X}_{S_i}, \mathcal{X}_{S_i})$. As $\Phi_{\mathcal{K}}(\cdot)$ helps us generalize the values of kernel gradient (and consecutively f_t, as it evolves according to kernel gradient) to values x outside the dataset S_i (and also p^{in}). Now that we have derived the closed form solution of the outputs of f_t on X_{S_i}, we can also derive the outputs of f_t on any arbitrary x by taking the integral of (24) and replacing f_t according to (25):

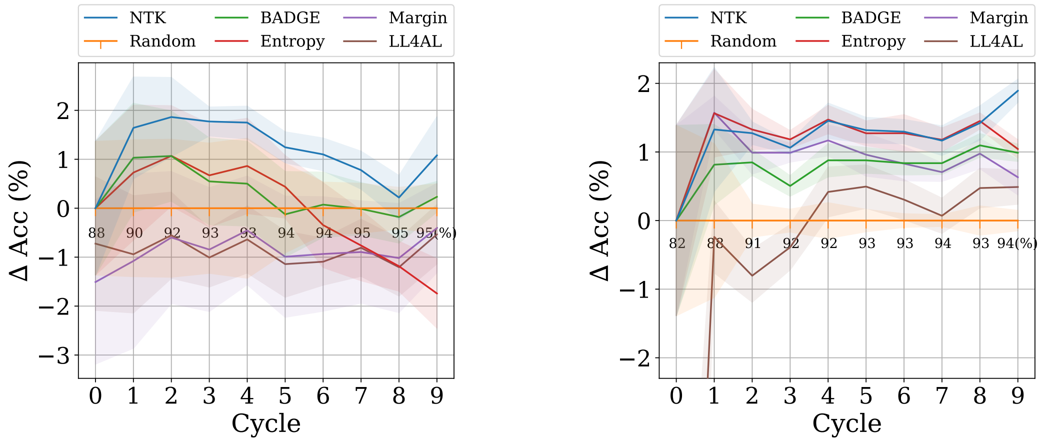
$$\begin{aligned}f_t(x) &= \frac{1}{2N}\sum_{x_j \in S_i} \mathcal{K}(x, x_j)\left[\int_{t'=0}^{t}\left(f_{t',z}(x_j) - f^*_{S_i,z}(x_j)\right) dt'\right]_z \\ &= f_0(x) + \mathcal{K}(x, \mathcal{X}_{S_i})\mathcal{K}_{S_i}^{-1}\left(I - e^{-t\mathcal{K}_{S_i}}\right)\left(f^*_{S_i}(\mathcal{X}_{S_i}) - f_0(\mathcal{X}_{S_i})\right)\end{aligned} \tag{26}$$

where $f_{\cdot,z}$ shows the zth index of output of f and $[\cdot]_z$ shows the stacked vector for different values of z, such that z is an integer in $[1 - n_L]$. Starting from initialization, we can use this to characterize the outputs of the trained neural network on S_i using gradient flow for time t. We're interested in starting from f_{S_i} and training on S_{i+1} for infinite time, according to the same loss function defined on p^{in}. For the sake of having more clear notations, we denote this function as $f_{S_i \to S_{i+1}}$ (defiend as $f_{S_i, S_{i+1}}$ in the statement that we are proving).

$$\begin{aligned}f_{S_i \to S_{i+1}}(x) &= f_{S_i}(x) + \mathcal{K}(x, \mathcal{X}_{S_{i+1}})\mathcal{K}_{S_{i+1}}^{-1}\left(f^*_{S_{i+1}}(\mathcal{X}_{S_i}) - f_{S_i}(\mathcal{X}_{S_{i+1}})\right) \\ &= \mathcal{K}(x, \mathcal{X}_{S_{i+1}})\mathcal{K}_{S_{i+1}}^{-1} f^*_{S_{i+1}}(\mathcal{X}_{S_{i+1}}) \\ &\quad + \underbrace{\left(\mathcal{K}(x, \mathcal{X}_{S_1}) - \mathcal{K}(x, \mathcal{X}_{S_{i+1}})\mathcal{K}_{S_{i+1}}^{-1}\mathcal{K}(\mathcal{X}_{S_{i+1}}, \mathcal{X}_{S_i})\right)}_{A}\mathcal{K}_{S_i}^{-1}\left(f^*_{S_i}(\mathcal{X}_{S_i}) - f_0(\mathcal{X}_{S_i})\right) \\ &\quad + f_0(x) - \mathcal{K}(x, \mathcal{X}_{S_{i+1}})\mathcal{K}_{S_{i+1}}^{-1} f_0(\mathcal{X}_{S_{i+1}}) \\ &= f_{S_{i+1}}(x) + A\mathcal{K}_{S_i}^{-1}\left(f^*_{S_i}(\mathcal{X}_{S_i}) - f_0(\mathcal{X}_{S_i})\right)\end{aligned} \tag{27}$$

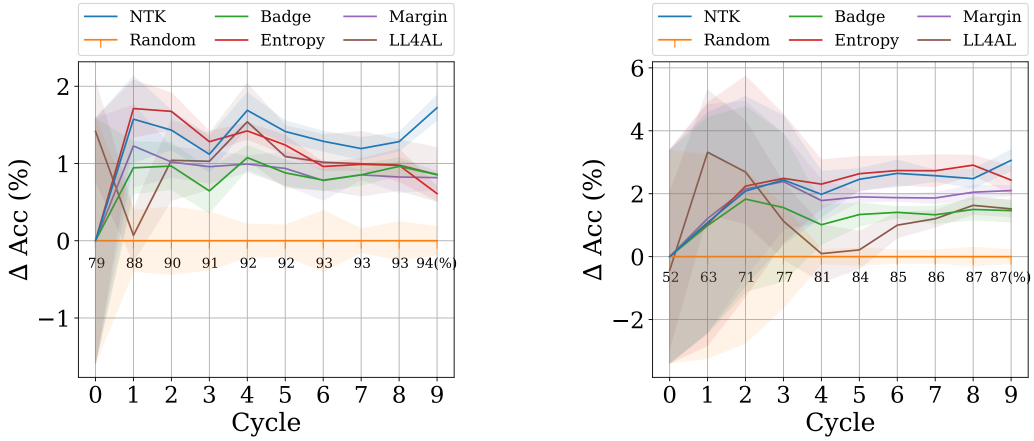
If we look closely, A can be written as:

$$\begin{aligned}A &= \mathcal{K}(x, \mathcal{X}_{S_{i+1}})\begin{bmatrix} I_{M_i} \\ O_{(M_{i+1}-M_i)\times M_i}\end{bmatrix} - \mathcal{K}(x, \mathcal{X}_{S_{i+1}})\mathcal{K}_{S_{i+1}}^{-1}\mathcal{K}(\mathcal{X}_{S_{i+1}}, \mathcal{X}_{S_i}) \\ &= \mathcal{K}(x, \mathcal{X}_{S_{i+}})\left(\begin{bmatrix} I_{M_i} \\ O_{(M_{i+1}-M_i)\times M_i}\end{bmatrix} - \mathcal{K}_{S_{i+1}}^{-1}\mathcal{K}(\mathcal{X}_{S_{i+1}}, \mathcal{X}_{S_i})\right) \\ &= \mathcal{K}(x, \mathcal{X}_{S_{i+1}})\left(\begin{bmatrix} I_{M_i} \\ O_{(M_{i+1}-M_i)\times M_i}\end{bmatrix} - \mathcal{K}_{S_{i+1}}^{-1}\mathcal{K}(\mathcal{X}_{S_{i+1}}, \mathcal{X}_{S_{i+1}})\begin{bmatrix} I_{M_i} \\ O_{(M_{i+1}-M_i)\times M_i}\end{bmatrix}\right) \\ &= \mathcal{K}(x, \mathcal{X}_{S_{i+1}}) \times 0 = 0\end{aligned} \tag{28}$$



(a) MNIST: 1-layer WideResNet

(b) SVHN: 2-layer WideResNet



(c) SVHN: ResNet18

(d) CIFAR10: ResNet18

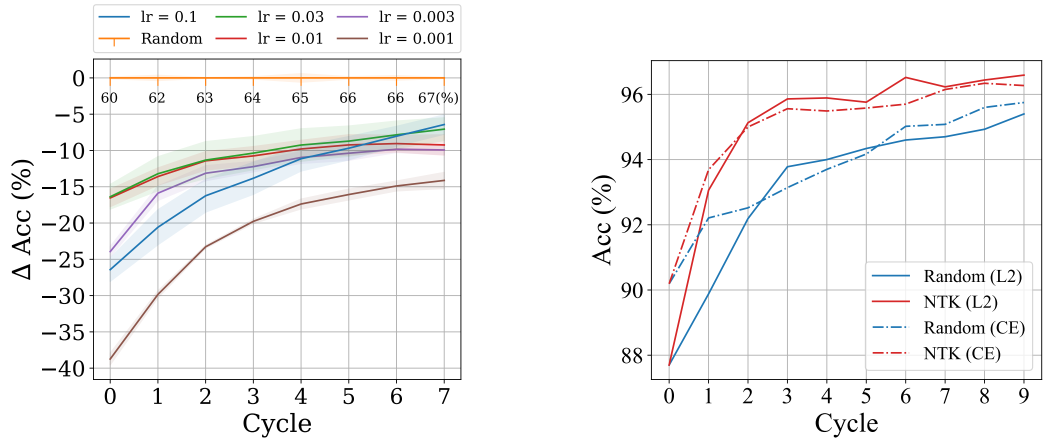
Figure 4: Comparison of the-state-of-the-art active learning methods on various benchmark datasets. Vertical axis shows difference from random acquisition, whose accuracy is shown in text.

where M_i denotes the number of datapoints in S_i. Note that (28) doesn't depend on the indices of S_i and S_{i+1}. Rather, it just relies on $S_i \subseteq S_{i+1}$, which is the case in Setting A for any consecutive batches. Thus, for any arbitrary x, $f_{S_i \to S_{i+1}}(x) = f_{S_{i+1}}(x)$. Using basic forward induction, starting from f_{S_i}, we can prove that $f_{S_1 \to S_2 \to \cdots \to S_C}(x) = f_{S_C}(x)$ for any arbitrary x. The proof is complete. □

B Additional Comparison with State-of-the-art Methods

In Figure 2, we demonstrate that our proposed method is equal or better compared to other state-of-the-art methods on various benchmark datasets in active learning, with different architectures. In this section, we provide additional experiment results as shown in Figure 2. Among all the combinations of the datasets we use (MNIST, SVHN, CIFAR10 and CIFAR100), and architectures (1-layer WideResNet, 2-layer WideResNet, and ResNet18), we do not provide results on MNIST with 2-layer WideResNet and ResNet18 since 1-layer WideResNet is large enough for MNIST. Also, we do not provide the results with 1-layer WideResNet on CIFAR10 since 1-layer WideResNet is too shallow for complicate data like CIFAR10 and 100. Similarly, according to our experiments, WideResNet with 1 or 2-layers are too shallow for CIFAR 100, unlike ResNet18.

As a result, in Figure 4, we provide the experiment results of all the other combinations. We visualize in the same way as we do for Figure 2 where we plot the difference between each method and random acquisition function at each cycle. Figure 4 shows that the proposed NTK approximation is equal or



(a) Varying LL4AL's learning rate (CIFAR100).

(b) L_2 and cross-entropy losses on MNIST.

Figure 5: Additional comparisons.

better than the other state-of-the-art methods; especially, NTK is better than any other methods for Figure 4a and Figure 4c, and comparable to entropy acquisition function for Figure 4b and Figure 4d.

C Instability of LL4AL

In Figure 2, we do not include LL4AL [38] performance on CIFAR10 or CIFAR100; its performance is too poor to show on the same plots. To validate this was not due to a simple poor hyperparameter choice, we show the performance of LL4AL on CIFAR100 with varying learning rates in Figure 5a. We run each experiment 3 times with the maximum learning rates of $\{0.3, 0.1, 0.03, 0.01, 0.003, 0.001\}$ for the 1cycle learning rate policy [47], which we use for all the other methods. When the maximum learning rate is 0.3, the gradient blows up and the model does not learn anything.

Figure 5a shows none of the learning rates are comparable with random acquisitions (trained with the maximum learning rate of 1.0), although it has been reported that LL4AL performs better than the random acquisition function. We conjecture this is because, if we train with random acquisitions using a learning rate that "works" for LL4AL, LL4AL performs better. On the other hand, when the learning rate is tuned to maximize the performance of each method, random acquisition can perform better.

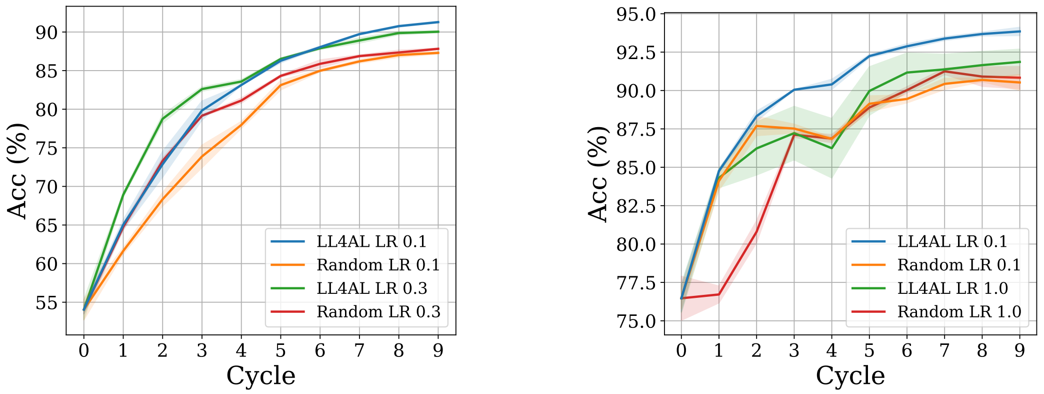
We added the figures Figure 6 based on the results shared in this GitHub repository which is a direct implementation of the LL4AL method. We further elaborate on each plot below:

CIFAR10, ResNet18 (Figure 6a): We observe that although when using the learning rate of 0.1 LL4AL's performance is 4% better than random (91.3% vs 87.3%), when the learning rate is tuned in favor of random (0.3 LR instead of 0.1) and use that for LL4AL too, the difference shrinks down to 1.2% (90.0% vs 87.8%). Moreover, using larger learning rates like 1.0 would result in LL4AL to diverge while random still gets descent results.

SVHN, ResNet18 (Figure 6b): Again, the difference is 3.3% (93.8% vs 90.5%) when LR is tuned in favor of LL4AL, but drops to 1.1% (91.9% vs 90.8%) when using a LR of 1.0. This time however, when LR is fully tuned in favor of random (2.0), LL4AL diverges.

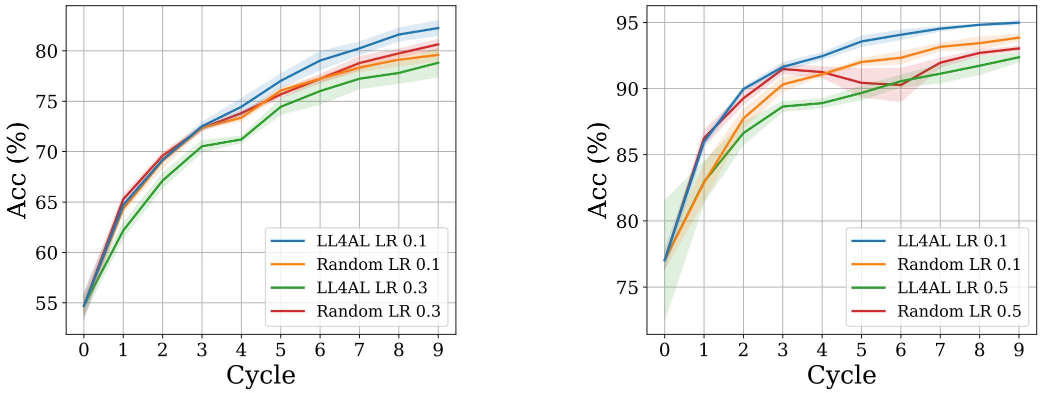
CIFAR10, 1-block WideResNet (Figure 6c): The 2.6% difference when using 0.1 LR (82.2% vs 79.6%) drops to -1.8% (78.8% vs 80.6%) when LR is tuned in favor of random (0.3). We remind that as we're using one block layer here, the LL4AL's performance drop is much more noticeable to the point that it performs worse than random when LR is tuned in favor of random!

SVHN, 1-block WideResNet (Figure 6d): The 2% difference when using 0.1 LR (95% vs 93%) drops to -1.4% (92.4% vs 93.8%) when LR is tuned in favour of random (0.5).



(a) Varying LL4AL's learning rate (CIFAR10 - ResNet18).

(b) Varying LL4AL's learning rate (SVHN - ResNet18).



(c) Varying LL4AL's learning rate (Cifar10 - 1-layer WideResNet).

(d) Varying LL4AL's learning rate (SVHN - 1-layer WideResNet).

Figure 6: Additional experiments on instability/poor performance of LL4AL on large learning rates.

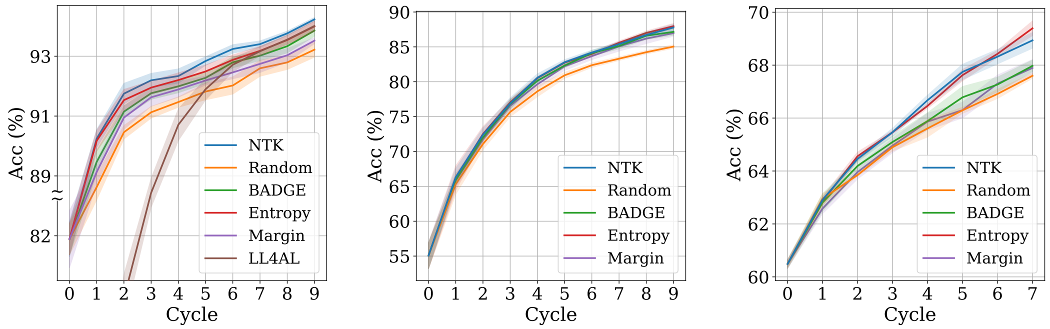
D Comparison of L_2 and Cross-entropy Loss

We use L_2 loss instead of cross-entropy loss to train a classifier. As mentioned in the main paper, Hui and Belkin [22] empirically show L_2 loss is just as effective as cross-entropy loss for various classification tasks in computer vision and natural language processing. Because of this, previous works that exploit a linearized network using empirical NTKs also use L_2 loss as a replacement for cross-entropy [25, 48].

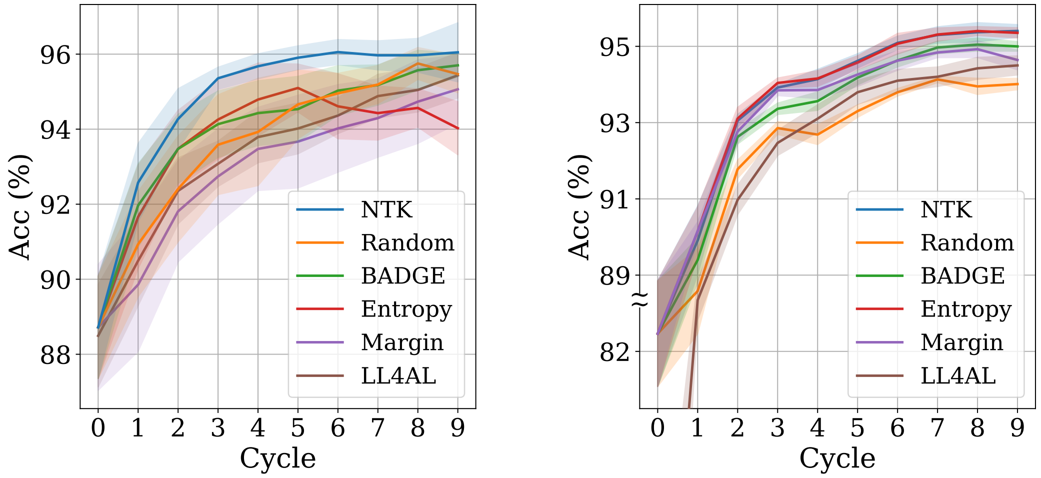
To further demonstrate that L_2 loss is indeed as effective as cross-entropy in active learning, we provide experimental results with L_2 and cross-entropy loss (CE) for both random and the proposed NTK acquisition functions in Figure 5b. Although cross-entropy starts with a higher accuracy, L_2 quickly catches up. Overall, the difference between L_2 and cross-entropy is not significant for either random or NTK acquisition functions.

E Conventional Format for Comparison of the Proposed Method vs. SOTA

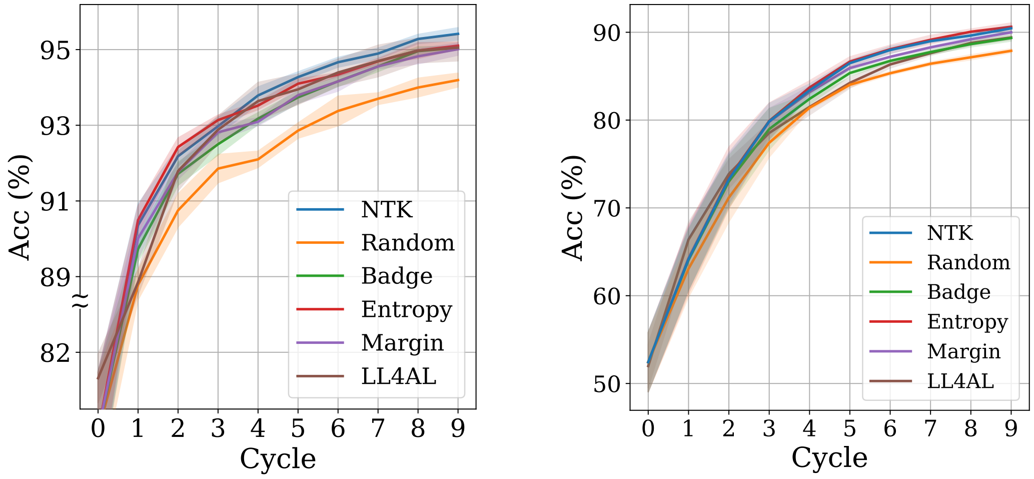
We have provided the conventional accuracy-based plots for the experimental results of comparing our proposed method with other state-of-the-art methods. We think the ΔAcc based plots are better at letting readers observe the differences between performances of each method on each task. However, for the sake of completeness and in case one finds the plots in the main paper confusing, we have provided the conventional plots in Figure 7.



(a) SVHN: 1-layer WideResNet (b) CIFAR10: 2-layer WideResNet (c) CIFAR100: ResNet18



(d) MNIST: 1-layer WideResNet (e) SVHN: 2-layer WideResNet



(f) SVHN: ResNet18 (g) CIFAR10: ResNet18

Figure 7: Comparison of the-state-of-the-art active learning methods on various benchmark datasets. Vertical axis shows attained accuracy of each acquisition method.